# 摘 要

虚拟化技术是云计算实现的关键技术之一，当前有着众多的虚拟化软件，同时也存在着众多的虚拟化管理软件。Libvirt 最为一种本地的虚拟化管理软件，能够管理诸如 Xen、QEMU、VMware 等众多的虚拟化软件工具。

为了使虚拟机在共享宿主机资源的时候能够对实现资源的互斥访问，libvirt 实现了一种锁管理框架。在此框架下可以使用各种能够在分布式上使用的建议锁来给出具体的实现，当前已经实现了三把锁：nop、sanlock 与 lockd。

本文通过论述这三把锁的优缺点以及 libvirt 当前锁机制的现状，分析锁管理框架的功能需求，确定了插件的总体结构设计，详细阐述了如何结合 Corosync 与 DLM 设计出一种更优的实现方案，并据此实现了 DLM 锁插件，并进行了相关的黑盒测试。

**关键字：虚拟化管理工具　libvirt　分布式锁管理器　DLM　Corosync**

# ABSTRACT

Virtualization technology is one of the key technologies for cloud computing. Currently, there are numerous hypervisors, and there are also hypervisor management tools. Libvirt is as the local hypervisor management software which could manage hypervisors such as Xen, QEMU, VMware, and so on.

In order to prevent multiple virtual machine from having concurrent write access to the same resources, libvirt implements a lock management framework where a variety of advisory that can be used on a distributed environment to give concrete implementations. Currently, three locks have been implemented: nop, sanlock, and lockd.

This thesis discusses the advantages and disadvantages of these three locks and the current status of libvirt's current lock mechanism, analyzes the functional requirements of the lock management framework, determines the overall structure design of the plugin-in, and elaborates on how to design a better implementation combining Corosync and DLM. Based on this, the DLM lock plug-in was implemented and some related black box tests were performed.

**Keywords：virtualization management tool　libvirt　distributed lock manager　Corosync　DLM**

# 目 录

# 第一章 绪论

## 1.1 研究背景和意义

随着云计算的兴起，越来越多的企业选择将计算业务运行在云平台之上。云计算实现的关键技术之一是虚拟化技术[1]，它能够在单个计算节点之上同时运行多个虚拟的实例，从而大大提高了计算资源的利用率，为用户提供了一个弹性的平台服务。此外，也可以利用虚拟化技术便捷地模拟软件运行的复杂环境，大大方便了软件的开发、调试、问题复现等诸多工作。当前存在着众多的虚拟化技术及其相应的软件实现，比如 KVM/QEMU、VirtualBox、VMware Workstation，Xen。如何在不同的运行环境下简单、方便地使用这些虚拟化软件管理虚拟机实例，这对虚拟化的管理软件提出了更高的要求。

在公有云的市场份额上，KVM/QEMU 占有着不低的比例，也是当前的趋势。它起初是一套由 Quramnet 公司开发的自由软件，支持全虚拟化(full virtualization)方案，对所模拟的虚拟机实例无需做出任何的修改即可直接运行。后来该公司于 2008 年被 Red Hat 收购，相应的软件也归 RedHat 所有。早在 2005 年，Red Hat 便为 Xen 专门设计了一种管理 API，称之为 libvirt，它后来被扩展为支持多个虚拟机监控程序[2]。随着 Red Hat 对 KVM/QEMU 的大力推广，libvirt 自然也成为 KVM/QEMU 管理软件的最佳选择，成为在向外扩展计算中最重要的库之一，作为云计算平台（比如 OpenStack）与 KVM/QEMU 对接的桥梁存在。

Libvirt 是一组开源软件的集合，包括三个部分：一个长期稳定的 C 语言应用程序接口（API）、一个守护进程（libvirtd）和一个命令行工具（virsh），主要目标是提供一个单一途径以管理多种不同虚拟化方案以及虚拟机[3]。在虚拟机之间，总存在无意中使多个虚拟机共同使用一个磁盘镜像的情况发生，如果在非集群文件系统下同时对磁盘进行写入，则可能导致发生不可逆转的数据损坏，而且当前并非每个虚拟化软件都能保证磁盘镜像的互斥使用。

Libvirt 中锁管理器（lock manager）框架的实现使得虚拟机在共享宿主机资源的时候能够实现对资源的互斥访问。典型的场景是实现一个写锁，阻止两个虚拟机

进程同时获取共享磁盘镜像的写入权限。这一技术的实现不仅可以保证多个KVM/QEMU 实例不同时使用一个磁盘镜像，同时也能够显式地标识所共享的磁盘[4]。它对保证磁盘镜像的互斥使用提供了一个新的解决方式。

本项目基于 libvirt 的锁管理器框架实现 DLM-Corosync 分布式锁管理器的设计，迎合了 DLM 与 Corosync 的应用场景，也通过对比当前的其他的实现说明 DLM-Corosync 锁管理器的优势所在。

## 1.2 国内外研究现状

Libvirt 分布式锁管理器框架作为保证数据可靠性的手段，从提出之初便探讨如何采用非硬编码方式嵌入 libvirt 的源码中，在具体的使用场景（比如分布式环境、动态迁移）下分析怎样更有效地保证互斥资源的使用。

自 libvirt 分布式锁管理器框架[5]的提出[6]到现在，相关的源码[7]经过多年的发展，形成了一个模块化的功能，在这个模块化的框架下可以很方便地实现采用不同类型的建议锁（Advisory locking）来负责锁管理器的具体实现。当前的操作系统中，存在多种类型的建议锁，比如通过 fcntl(F_SET_LK)使用的标准 POSIX 文件锁（Standard POSIX File locking），oVirt 项目实现并推荐的 sanlock，还有常用于高可用集群（High Availability Clustering）中且在 Linux 内核实现的分布式锁管理器（Distributed Lock Manager，DLM），等等。

基于 libvirt 的特点，Daniel P. Berrange 在 libvirt 分布式锁管理器框架下实现了 nop[8]、sanlock[9]与 lockd[10]。前者什么都不会做，它在当 libvirt 在不需要任何类型锁的时候非常有用，后两者能够保证集群中的共享资源互斥使用。Sanlock 是一个基于 SAN 的分布式锁管理器，集群中的每个节点都各自运行着 sanlock 服务，锁的状态都被写到了共享存储上，使用 disk-paxos 算法读写共享存储以实现对分布式锁的获取、释放和超时[11]。但 sanlock 的使用会带来大量的磁盘 I/O 操作，因此又提出了 lockd 这种使用标准 POSIX 文件锁的实现[10]。大部分分布式文件系统（Distributed File System）都遵循 POSIX 标准实现了 POSIX 文件锁（比如 NFS），这就为 lockd 的实现奠定了基础。在具体的实现上，libvirt 使用了 virtlockd 守护进程进行锁资源的管理，以便 libvirt 守护进程重启后仍然能够对锁继续持有。当前的

libvirt 版本下，默认使用 nop，采用 lockd 作为被注释掉的 lock_manager 默认配置选项。

这种类型的数据损坏仍然还在发生[14]。这是因为有些独立于 libvirt 之外工具（比如 qemu-img）的使用并不能检测到磁盘镜像是否被别的进程使用，因此 QEMU 引入了 OFD 文件锁（open file locking）这项特性[12][13]。这项特性引起了广泛的讨论，对此，libvirt 社区中有成员期望着使用其他机制的锁管理器插件的出现。

## 1.3 主要研究内容

本项目针对传统高可用集群下的运行环境情况，利用集群中已经存在的 DLM 和 Corosync 软件，在 libvirt 分布式锁管理器框架下给出另一种锁 DLM 的设计与实现。

项目旨在通过学习 DLM 与 Corosync 的原理、基本理论与技术，掌握 DLM 和 Corosync 的使用方法与高可用集群的部署，了解 libvirt 的使用方式与内部实现，在锁管理器框架下设计一个能够对共享资源进行保护的插件，并给出相关的黑盒测试方案证明这种实现是可行的。

本人在项目中涉及到的工作主要有：

- 查找文献和资料，学习虚拟化系列软件 QEMU、libvirt 以及 virt-manager 的使用，掌握 Linux 的网络配置方法，巩固 C 语言和 Linux 环境编程能力。
- 学习高可用集群的部署方法，了解 DLM 与 Corosync 的使用方法与功能。
- 分析 libvirt、DLM 与 Corosync 源码，了解大致的工作流程，掌握 libvirt 锁管理框架的实现细节与作用点，以及 sanlock 与 virtlock 的工作原理。
- 进行插件的需求分析和设计，使用 C 语言实现插件。
- 基于 DLM 与 Corosync、以及 NFS 部署一个简易的集群环境，完成测试环境的搭建并进行测试。

## 1.4 论文组织结构

本论文共分为六章，具体内容如下：

第一章，绪论。本章主要介绍了项目背景和意义，对虚拟化管理工具 libvirt 及其锁管理框架的国内外研究现状做了简要介绍与分析，阐明了本论文的具体内容展开和主要工作。

第二章，系统关键技术。本章主要介绍了项目用到的 DLM 与 Corosync 的理论基础，以及对相应的软件环境技术：高可用技术、虚拟化技术，和相应的软件 libvirt 等进行了简要介绍。

第三章，插件需求分析与设计。本章主要通过应用环境对插件进行了需求分析，之后通过数据流图和数据字典简要阐述了插件的功能设计。

第四章，插件的设计与实现。本章介绍了插件的总体设计与详细设计，通过插件的数据结构、初始化模块、资源关联模块与锁的操作模块四部分，辅以流程图详细阐述了插件的功能设计与实现。

第五章，插件的测试与分析。本章给出了测试目标与黑盒测试方案：单机环境下的测试与集群环境下的测试，用以证明该插件能够以预期的方式正确地工作。

第六章，总结与展望。本章是论文的结束章节，主要对论文进行总结，分析取得的成果与不足，并对进一步工作提出展望。

# 第二章 系统关键技术

## 2.1 高可用技术与 DLM、Corosync

### 2.1.1 高可用技术概述

随着 Internet 技术的迅猛发展，各项服务逐渐通过网络技术来提供，人们已习惯于网络带来的便捷和高效率。但是计算机系统也非常脆弱，它会受各种因素的影响，这对系统的可用性提出了越来越高的要求。为了尽可能地减少宕机风险，各种平台下的高可用技术方案应运而生。高可用技术是一种抵御指系统中的不确定性，保证系统无中断地执行其功能的技术，能够实现高度可用的物理和虚拟集群，并排除故障点，确保网络资源包括数据、应用程序和服务的高可用性和可管理性。高可用软件套件能够提供必需的监视、消息交换和集群资源管理功能，支持对独立管理的集群资源进行故障转移、故障回复和迁移（负载均衡）。

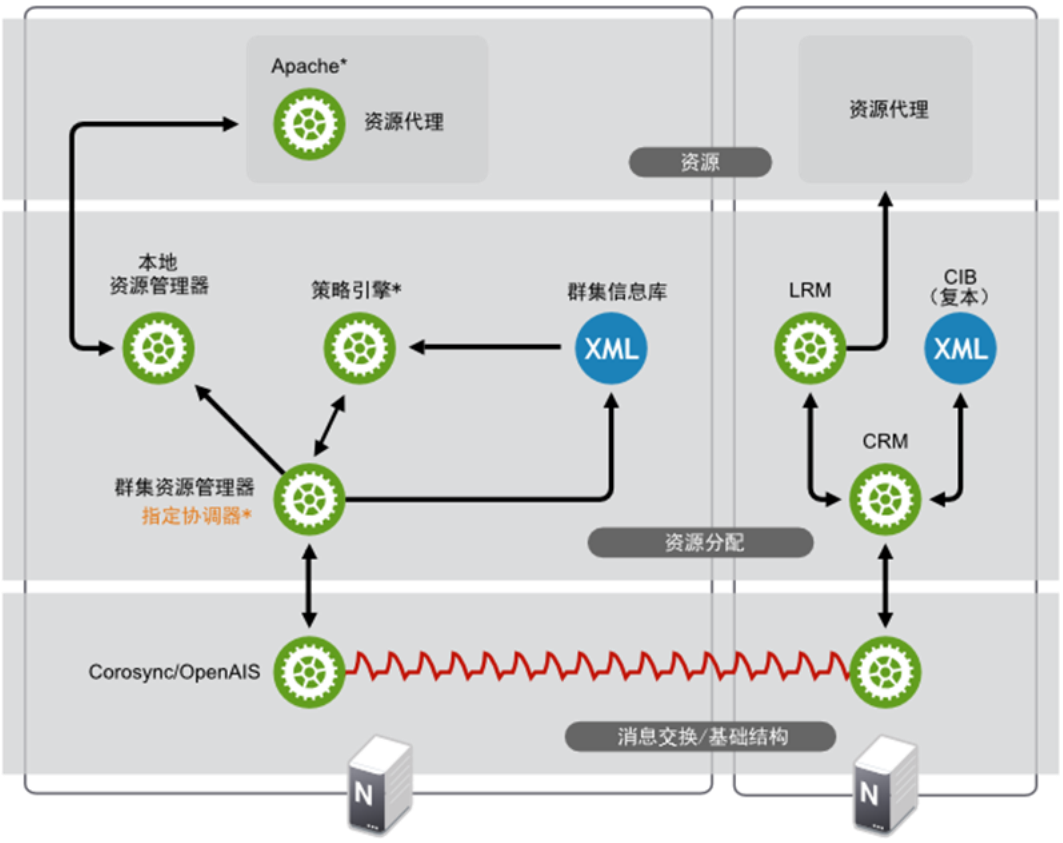


图 2-1 高可用套件的体系结构

图 2-1 是一种开源的高可用套件体系结构。此图描述了高可用套件下软件成员的层次结构以及与此相关的组件：主层，或者第一层是消息交换和基础结构层，也称为 Corosync/OpenAIS 层，此层包含了发送含有“我在线”信号的消息及其他信息的组建，是集群间节点通信的基础；下一层是资源分配层，包含群资源管理器 CRM、群集信息库 CIB、指定协调器 DC、策略引擎 PE 和本地资源管理器 LRM，这些组件能够在集群中监控并汇报资源信息、协调节点之间的资源分配、维护集群成员资格；最高层是资源层，资源层包括一个或多个资源代理 RA。资源代理是已写入的用来启动、停止和监视某种服务（资源）的程序[15]。

### 2.1.2 Corosync 概述

Corosync 全称 The Corosync Cluster Engine，它是集群节点之间通信的基础设施。项目基于以下的历史原因创建：因为意识到了集群设施迫切需要拥有属于自己的项目名称和社区，因此创建者借鉴了 OpenAIS 项目以及 Pacemaker 与 Apache Qpid 的工作经验，为 Corosync 设计了灵活、紧凑的架构，使之可以方便地应用于多种集群套件之中。之后，诸如 OCFS2 和 GFS 文件系统的集成层，以及 Pacemaker 和 Apache Qpid 等其他项目进一步扩展了 Corosync 的通用性，不仅使之能够更好地与不同的软件协同使用，而且提高了它的代码质量[16]。

它是一个具有集群通信功能的系统，并具有实现应用程序内高可用性的特点。Corosync 提供了四种类型的 C 语言应用程序接口：

- 一个具有虚拟同步功能的封闭的进程组通信模型，可用于创建复制状态机。
- 一个简单的可用性管理器，可在应用程序退出后重新启动它。
- 一个配置和统计的内存数据库，拥有设置、检索和接受信息更改通知能力。
- 一个可以告知法定人数是否满足要求投票系统。

### 2.1.3 DLM 概述

DLM 全称 Distributed Lock Manager，是一种建议性锁，用以协调共享资源的使用，从而允许在集群中的多个节点上并发地运行程序。协作的程序可以运行在集群的不同节点上，共享一个资源而不会导致它损坏。Linux 内核实现了 DLM，并

在诸如 OCFS2 等的集群文件系统中获得了广泛的使用[17]。

DLM 锁具有丰富的种类，以便在不同的场景下使用。除此之外，它还包括同步和异步这两种运行方式。这些特性包括：

- 六种锁类型，有着越来越严格的资源访问限制。
- 具有锁类型的转换操作，可以用于提升降低对资源的访问限制。
- 同步完成锁操作。
- 异步完成锁操作。
- 利用锁值块保存全局变量。

锁模式有六种，分别是：NL、CR、CW、PR、PW、EX。他们的兼容矩阵如表 2-1 所示：

表 2-1 锁模式的兼容性

| **请求的锁** | **当前已授予的锁** | | | | | |
|---|---|---|---|---|---|---|
| | NL | CR | CW | PR | PW | EX |
| NL | Yes | Yes | Yes | Yes | Yes | Yes |
| CR | Yes | Yes | Yes | Yes | Yes | No |
| CW | Yes | Yes | Yes | No | No | No |
| PR | Yes | Yes | No | Yes | No | No |
| PW | Yes | Yes | No | No | No | No |
| EX | Yes | No | No | No | No | No |

DLM 将锁资源（lock resource）定义为可锁定的实体。DLM 会创建一个锁资源以响应该锁资源的第一个锁请求。当与锁资源关联的最后一个锁被释放后，DLM 便会销毁这个锁资源。一个锁资源可以有多个锁与之相关联，而锁始终只与一个锁资源相关联。

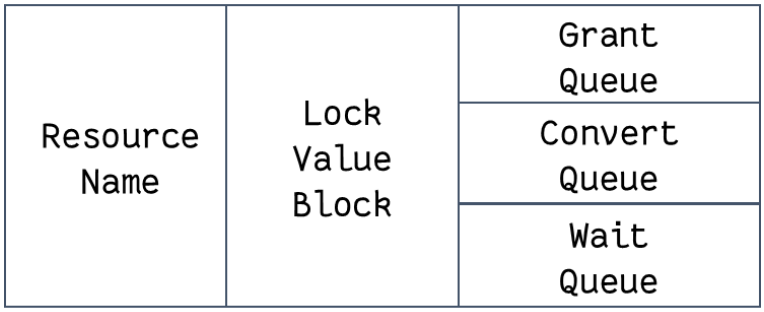


图 2-2 锁资源与锁

图 2-2 是 DLM 中的锁资源的抽象描述，从图中可以看到，它的组成包含三个部分：名字、锁值块与锁队列集合。相关解释如下：

- 名字。一个不能超过 64 字节的字符串。
- 锁值块。具有 32 字节的存储空间，可用于存储应用程序的数据。
- 锁队列集合。每个锁资源都与三个队列相关联，他们中每一个都是可能的锁状态。授予队列包含 DLM 授予的所有锁，但不包括那些正在转换的锁。转换队列是那些已被授予、但正在尝试转换为另一种模式的锁，但这种锁模式与某些已授予的锁相冲突。等待队列包含尚未被授予的新锁请求。DLM 都按照先入先出的顺序处理转换队列与等待队列中的锁。

DLM 提供了一个可在集群间共享的单一且统一的锁映像，每个节点都运行着 DLM 内核守护进程的副本，这些 DLM 守护进程相互通信，共同维护者集群范围的锁资源数据库和这些资源上的锁。

在此集群范围内的数据库，DLM 维护着每个锁资源的一个主副本(one master copy of each lock resource)。这个主副本可以驻留在集群的任何节点上。最初，主副本驻留在发起锁请求的节点上。DLM 维护着集群范围内的地址目录，它上面记载着所有节点中锁资源的主副本。DLM 试图在集群节点中平均分配目录。当程序向某个锁资源请求加锁时，DLM 首先确定哪个节点保存着目录条目，之后读取目录内容，找出当前哪个节点保存着锁资源的主副本。允许所有节点维护着锁资源的主副本，而不是在集群中只由一个 DLM 来做这个事情，这样锁请求便可以在本地节点上被处理，从而减少网络流量。锁请求的本地处理也避免了潜在的网络瓶颈，避免了当 DLM 故障时重建锁数据库锁带来的时间开销。使用这些技术，DLM 具有更高的锁吞吐量以及更低的网络流量开销。

当节点发生故障时，在运行正常的集群节点上的 DLM 会释放由故障节点所持有的锁。之后处理来自运行正常的节点在故障发生之前发出的锁请求，而故障节点的锁请求会被阻止。此外，故障节点的锁将被其他节点重新持有。

DLM 提供了自己的策略来支持它的锁的特性，有着自己的内部通信协议，能够管理锁并且在节点发生故障后经过一系列恢复操作后还能够保持集群的正常运行，以及当新的节点加入集群后迁移锁。但 DLM 并不提供管理集群本身的机制，

若期望 DLM 在任何集群中正常工作，集群基础设施环境需要满足以上最低要求：

- 节点活跃度(node liveness)：节点是集群的一部分。
- 一致的成员视图(consistent view of membership)：所有节点都同一集群的成员资格。
- IP 地址活跃性(IP address liveness)：DLM 用于在节点间通信的 IP 地址；一般情况下 DLM 使用 TCP/IP 进行节点间通信，将每个节点限制为单个 IP 地址(尽管可以绑定多个地址来增加冗余度)。DLM 也可以配置为使用 STCP 协议，这允许每个节点拥有多个 IP 地址。

在 Linux 发行版上，DLM 提供了一个由 C 语言实现的名为 libdlm 的库，由它提供了相关的应用程序接口集合，允许用户空间的程序直接获取、操作和释放 DLM 锁，使 DLM 更易于使用。下面列出会在该项目中用到的一些 API，并给与简单的说明：

- dlm_lshandle_t dlm_create_lockspace(const char *name, mode_t mode)
  创建一个名为 name 的锁空间，并且该空间的访问权限为 mode。该锁空间在集群节点中必须不存在，否则会返回-1，并设置 errno 为 EEXIST，此时可以使用 dlm_open_lockspace 函数打开。调用成功后名为 name 的杂项设备会在/dev/misc 目录下创建。若创建成功，与此锁空间有关的句柄将会被返回，用于 dlm_ls_lock/unlock 的操作。
- dlm_lshandle_t dlm_open_lockspace(const char *name)
  打开一个已经存在的名为 name 锁空间，调用成功后返回与此锁空间操作相关联的句柄。
- int dlm_close_lockspace(dlm_lshandle_t ls)
  关闭一个锁空间。关闭成功之后，任何存在于此锁空间中的锁将会被释放，除非锁标志中设有 LKF_PERSISTENT。若存在与此锁空间相关联的线程，则该线程会被终止。
- int dlm_ls_pthread_init(dlm_lshandle_t lockspace)
  创建作用于 lockspace 锁空间上 AST 线程，它用于执行锁操作的异步通知。与锁操作有关的 AST 回调函数将会在该线程的上下文中被调用。

- int dlm_ls_purge(dlm_lshandle_t lockspace, int nodeid, int pid)

  释放所有锁。这些锁都是孤儿锁，位于 nodeid 节点上，并且存在于名为 lockspace 锁空间中，且与与特定进程 pid 相关联。若 pid 为 0，则代表 lockspace 中的所有孤儿锁。

- int dlm_ls_lockx(dlm_lshandle_t lockspace,
  uint32_t mode,
  struct dlm_lksb *lksb,
  uint32_t flags,
  const void *name,
  unsigned int namelen,
  uint32_t parent,
  void (*astaddr) (void *astarg),
  void *astarg,
  void (*bastaddr) (void *bastarg),
  uint64_t *xid,
  uint64_t *timeout)

  申请锁。它是带有超时参数的异步版本。如果申请锁成功，则会返回 0，并且通过 AST 例程为结构体 lksb 中的成员设置相应的参数。如果申请锁失败，则会返回-1，AST 例程要么不会被调用，要么 AST 例程被调用但 lksb 中的成员 sb_status 被设置为非 0。Lockspace 是锁空间的句柄；mode 是申请或要转换的锁的模式；flags 是与此请求有关的标志；lksb 为锁状态的存储区域，其成员用于存放申请锁的结果和相应的锁信息；name 为锁的名称；namelen 为 name 的字符串长度；parent 为 NULL，当前并未使用 astaddr 为 AST 例程的函数指针；astarg 为 ast 例程的参数；bastaddr 为请求阻塞时调用的 AST 例程的函数指针；bastarg 为 bast 例程的参数；xid 为有关死锁探测的事务标识；timeout 为超时时间。如果锁请求超时 AST 会触发 ETIMEOUT 状态，此时，若该锁处于转换状态则会保留转换之前的状态，否则该锁不再存在。未设置 LKF_TIMEOUT 标志超时参数会被忽略。

- int dlm_ls_lock_wait(dlm_lshandle_t lockspace,
  uint32_t mode,
  struct dlm_lksb *lksb,
  uint32_t flags,
  const void *name,
  unsigned int namelen,
  uint32_t parent,
  void *bastarg,
  void (*bastaddr) (void *bastarg),
  void *range)

  申请锁的同步版本。

- int dlm_ls_unlock_wait(dlm_lshandle_t lockspace, uint32_t flags, uint32_t lkid, struct dlm_lksb *lksb)

  请求释放的同步版本。

## 2.2 虚拟化技术 KVM 与 QEMU、libvirt

### 2.2.1 虚拟化技术 KVM 概述

KVM 全称 Kernel-based Virtual Machine，是一种用于 Linux 内核中的虚拟化基础设施，可以将 Linux 内核转化为一个虚拟机监视器（Hypervisor），在具备 Intel VT 或 AMD-V 功能的 x86 平台上运行，后来也被移植到其他平台之上，是全虚拟化的一种，允许未经修改的客户操作系统隔离运行。它包含一个为处理器提供底层虚拟化的核心模块 kvm.ko（kvm-intel.ko 或 kvm-AMD.ko），还需要一个软件 QEMU 作为虚拟机的上层控制和界面，提供虚拟的外部设备。KVM 本身并不模拟硬件，仅仅暴露出/dev/kvm 接口用于设置客户机的地址空间，提供客户机所用的虚拟 I/O，以及将客户机的视频显示映射回宿主机，关注虚拟机的调度、内存管理。QEMU 本身可以采用二进制指令翻译的方式运行虚拟机，为了使虚拟机的运行效率更高，它于 0.10.1 版本开始增添了对 KVM 模块的支持。

图 2.3 是 KVM/QEMU 虚拟化环境的一个宏观的描述[18]：

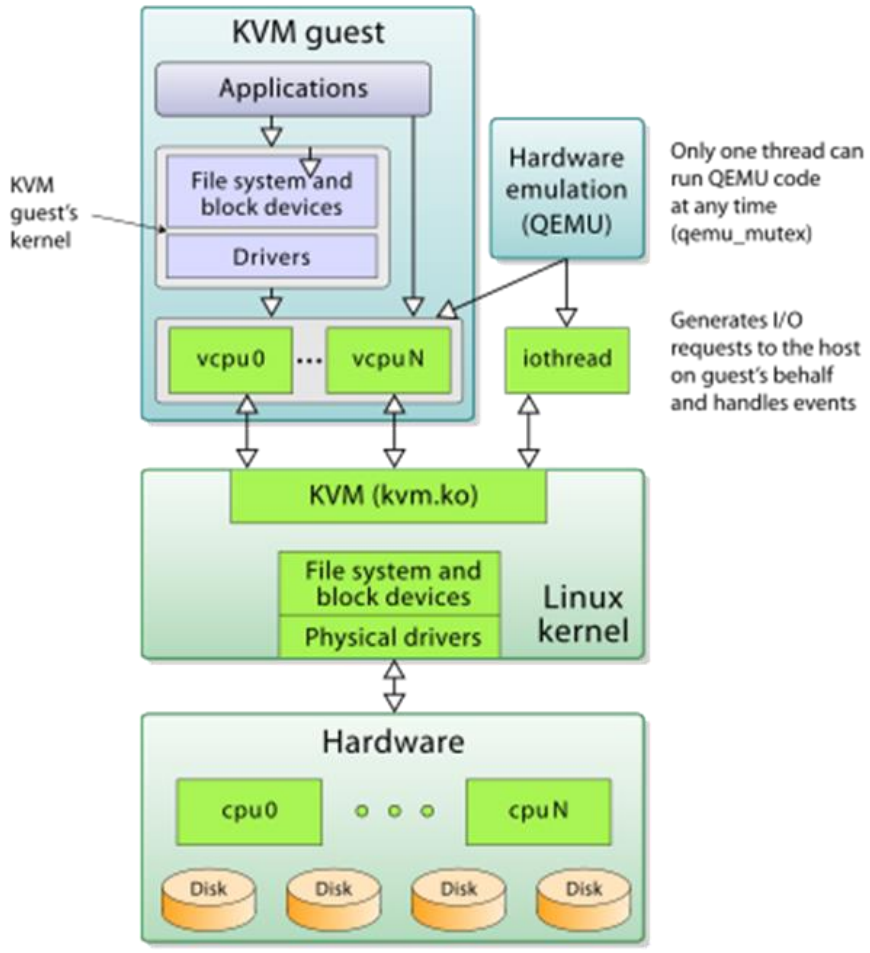


图 2-3 KVM/QEMU 虚拟化环境

### 2.2.2 Libvirt

#### 2.2.2.1 libvirt 概述

libvirt 是 libvirt 项目组的核心软件，它主要是由 C 语言写成，整个项目致力于提供在不同的操作系统平台之上以单一的方式管理不同的虚拟化方案以及虚拟机，同时提供 C、Python、Perl、Java 等多种计算机语言的应用程序接口，以便在在不同的计算机程序语言中方便地使用它。当前，它已被多个与云计算有关的项目所使用，比如 OpenStack，k8s（Kubernetes）。

图 2-4 形象地描述了 libvirt 在一些虚拟化套件中所起到的作用[19]：

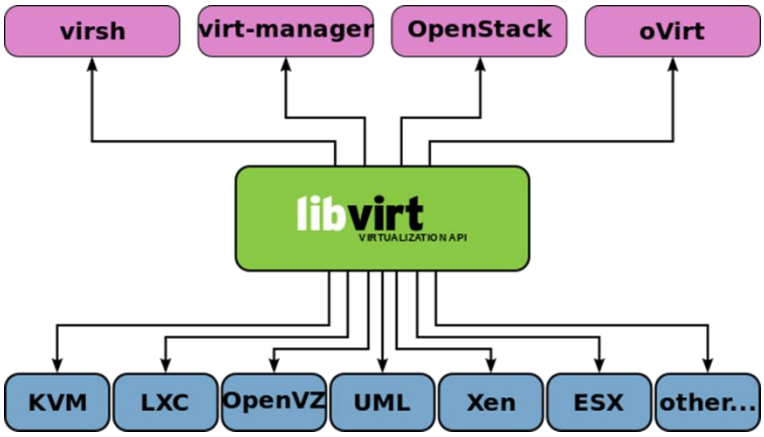


图 2-4 libvirt 与它支持的虚拟化软件及其使用它的虚拟化管理解决方案

#### 2.2.2.2 libvirt 虚拟机的生命周期

Libvirt 使用 domain 来描述一个运行在虚拟机器上的操作系统实例，它可以是一个运行着的虚拟机，或者用于启动虚拟机的配置。Guest domain 是用 XML 配置来描述的。Libvirt 使用 XML 文件格式来保存它的所有对象的配置，包括网络、存储、内存和其他元素，易于使用编辑器编辑这些 XML 配置[20][21]。

Libvirt 区分两种不同类型的 domain：短暂性的和持久性的。短暂性的 domain 只会在 domain 被关机或者所在的主机被重启之前存在；持久性的 domain 会一直存在，直到被删除。无论是什么类型的 domai n，它被创建后的状态可以被保存进一个文件，因此即使是一个短暂性的 domain 也可以被反复地恢复。

一个 Guest domain 可能处于以下几种状态的任意一种：

- Undefined（未定义的）：这是起始状态。这时 libvirt 不会直到 domain 的任何信息，因为这时候 domain 尚未被定义或者创建。
- Defined（定义了的）/Stopped（停止的）：domain 已经被定义，但是不在运行/只有持久性的 domains 才能处于该状态。当一个短暂性 domain 被停止或者关机时，它就不存在了。
- Running（运行中的）：domain 被创建而且启动了，无论是短暂性 domain 还是持久性 domain。任何处于改状态的 domain 都已经在主机的 hypervisor 中被执行了。
- Paused（中止了的）：Hypervisor 上对该 domain 的运行被暂替了。他的状态被临时保存（比如到内存中），直到它被继续（resumed）。Domain 本身

不知道它处于是否被中止状态。

- Saved（保存了的）：类似中止状态，除了 domain 的状态被保存在诸如硬盘之类的持久性存储上。处于该状态上的 domain 可以被恢复。

图 2-5 描述了 domain 的状态机（方框表示状态，箭头表示使得状态变更的命令）。从状态图中可以看出，对持久性 domain，shutdown 命令可以将其从 running 状态编程 undefined 状态；对短暂性 domain 而言，它会从 running 状态变为 undefined 状态。

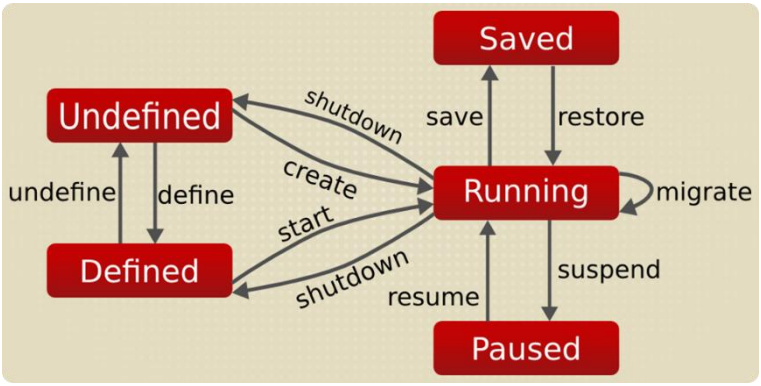


图 2-5 libvirt 中的 domain 状态机

一个运行中的 domain 或者虚拟机可以被迁移到另一个主机上，只要虚拟机的存储是在主机之间共享的，并且目标主机的 CPU 能够支持虚拟机的 CPU 模式。根据类型和应用，虚拟机迁移可以不导致虚机上运行的服务的中断。Libvirt 支持多种适用于不同场景下的迁移模式：Standard（标准模式）、Live（实时模式）、Non-live（非实时模式）、Peer-to-peer（对等模式）、Tunnelled（隧道模式）、Direct（直接模式）。

Libvirt 支持快照功能。一个快照是虚拟机的操作系统和它的所有应用在某个时刻的视图。在虚拟化领域，提供可以被恢复的虚机快照是个非常基本的功能。快照允许用户保存虚机在某个时刻的状态，然后在将来某个时候回滚到该状态。基本的用例包括：创建快照、安装新的应用、更新或者升级，然后回滚到之前的某个时间点。显然，在快照生成之后发生的任何操作都不会包含在快照中。并且，一个快照不会持续更新，它只表示虚机在某个特定时间上的状态。

2.2.2.3 libvirt 管理

绝大部分的 libvirt 管理可以通过三个工具实现：图形界面下的 virt-manager、交互式终端下的命令行工具 virsh 以及专门用于管理虚拟机磁盘镜像的 guestfish（它是 libguestfs 项目中的一部分）。

Virsh 工作在交互式的终端管理环境中，用于管理 Guest domain。因为需要通过通信管道与运行在超级管理员模式下的程序 libvirtd 守护进程通信，因此绝大部分 virsh 命令需要超级管理员权限来执行。Virsh 下的命令允许直接以附加命令行选项参数的方式执行，也可以在进入其内置的交互式终端中直接运行相应的命令。下面列出一些命令实例，绝大部分在第五章中的黑盒测试中有所使用：

- 连接本地的 libvirtd 守护进程

  # virsh --connect qemu:///system
- 查看当前 libvirtd 管理下的所有虚拟机状态

  # virsh list --all
- 启动、创建一个名为<domain>的虚拟机

  # virsh start <domain>

  # virsh create <domain>
- 正常关闭、强制关闭名为<domain>的虚拟机

  # virsh shutdown <domain>

  # virsh destroy <domain>
- 暂停名为<domain>的虚拟机

  # virsh suspend <domain>
- 唤醒名为<domain>的虚拟机

  # virsh resume <domain>
- 保存名为<domain>的虚拟机

  # virsh save <domain>
- 恢复名为<domain>的虚拟机

  # virsh restore <domain>
- 在线迁移虚拟机<domain>

# virsh migrate <domain> --live <uri> --unsafe

- 创建虚拟机<domain>的快照

  # virsh snapshot-create-as <domain> snapshot1 --disk-only –atomic

- 查看虚拟机<domain>的快照

  # virsh snapshot-list <domain>

- 编辑虚拟机<domain>配置文件

  # virsh edit <domain>

## 2.3 本章小结

本章主要对项目中使用到的相关技术和理论进行了简单的介绍：分析了高可用技术产生的背景及一种开源的高可用套件体系结构，阐述了 DLM 和 Corosync 技术细节，还介绍了虚拟化技术、KVM/QEMU，以及简要概述了开发 DLM-Corosync 所用到的 libvirt。

# 第三章　插件需求分析与设计

## 3.1　插件需求分析

本项目利用 DLM 建议锁与 Corosync 基于 libvirt 分布式锁管理器框架实现了名为 dlm 的分布式锁管理器插件，它可以用于阻止多个 QEMU 实例在相同的时刻下使用同一个磁盘镜像，除非这个磁盘镜像被显性地标识为共享的或者只读的。它旨在以下场景中实现资源互斥[22]：

- 两个不同的虚拟机使用同一个磁盘镜像。
- 因管理员的失误导致的同一个虚拟机在不同的宿主机上的多次启动。
- 因 libvirt 的驱动错误导致的一个虚拟机在同一个宿主机之上被多次启动。

为了满足上述目标，分布式锁管理器的实现必须满足以下的几点要求：

- QEMU 进程打开磁盘的时候，这个磁盘镜像必须被加锁。
- 插件必须实现只读、共享写、互斥写三种锁的模式。
- 在 QEMU 实例迁移的过程中，锁的所有权转移必须可行。
- 如果某个资源的锁被移除，锁管理器必须能够识别并杀死使用这个资源的进程实例。
- 通过使用管理程序可以对与虚拟机有关的任意资源进行加锁。

因此需要设计以下几种操作的实现来支持锁管理器达成上述目标：

- Register object：注册一个用于获取锁的对象，该对象的标识符是独一无二的。
- Add resource：把锁的对象与资源相互关联，用于之后对锁的获取/释放操作。
- Acquire locks：为与锁的对象相关联的所有资源进行加锁。
- Release locks：释放与锁的对象相关联的所有资源的锁。
- Inquire locks：查询与该锁的对象所关联的锁的状态。

同时，资源是虚拟机的一部分，一个虚拟机在宿主机上表现为与 libvirt 互相独立的一组进程或者一组线程。Libvirtd 守护进程的退出、重启并不会影响 QEMU 虚

拟机进程实例，除非宿主机被重启。因此还需要设计一种策略，用以保证 libvirtd 守护进程重启后能够重启获取这些与资源相关联的锁：

- 锁与相关联的虚拟机进程组有着相同的生命周期。
- 锁不会因 libvirtd 守护进程的重启、退出而失效。

综上所述，设计出的 libvirt 分布式锁管理器插件的模块功能结构图如下图 3-1 所示：

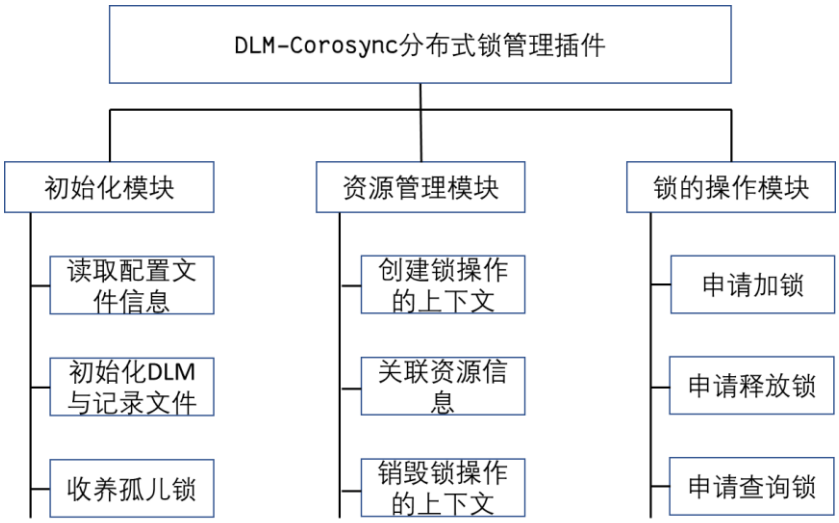


图 3-1 libvirt 分布式锁管理器插件模块功能结构图

## 3.2 插件功能设计

首先定义一个名词：孤儿锁。它指的是：因为锁的持久化属性的存在导致其本身的生命周期独立于 libvirtd 守护进程，并不受其生命周期的影响，因此锁只能被其拥有者释放，即谁拥有谁释放。在拥有者进程退出后，又没有新的进程通过回收操作成为锁的拥有者之前，该锁称之为孤儿锁。

通过使用 DLM API 申请锁的时候，可以通过添加参数 LKF_PERSISTENT 来用于将要申请的锁持久化，以便令 libvirtd 守护进程的退出或重启之后不会造成锁的自动释放。若宿主机没有重新启动，且相应的 QEMU 实例还在运行中，则说明与锁相关联的资源仍然在被使用，那么在 libvird 守护进程重新启动的时后便需要对孤儿锁进行收养；若宿主机有被强制重启的情况出现，在这种情况下 QEMU 实例自然也不复存在，除非它被迁移到别的宿主机之上，这时候与锁相关联的资源不

再被使用，Corosync 与 DLM 组成的分布式集群系统能够很好地处理这种情况——孤儿锁在集群中的记录自动被抹除，此节点上的锁空间不再存在，也就无需进行收养操作。

通过以上的需求分析，libvirt 分布式锁管理器插件主要分为三个功能模块，分别是：初始化模块、资源关联模块和锁的操作模块。初始化模块用于初始化分布式锁管理器插件，如果孤儿锁存在则收养孤儿锁。资源管理模块采用了一种策略，令每一个锁与特定的资源相互关联，达到一一映射的目的，确保锁与资源的所有者进程组有着相同的生命周期。锁的操作模块包含申请锁、释放锁与查询锁三个功能，用于控制与锁关联资源的访问。

### 3.2.1 初始化模块

初始化模块提供了插件的初始化操作。鉴于 DLM 与 Corosync 的特性，在模块初始化的时候，需要读取相应的配置文件信息，对不存在的配置信息采用默认的配置，之后对 DLM 与 Corosync 以及锁管理器插件进行初始化操作，最后收养可能存在的孤儿锁。

DLM 有锁空间（lockspace）的概念，类似于编程语言中的命名空间（namespace），用于在集群范围内唯一地标识一组锁的集合。

初始化模块的数据流图 3-2 如下所示：

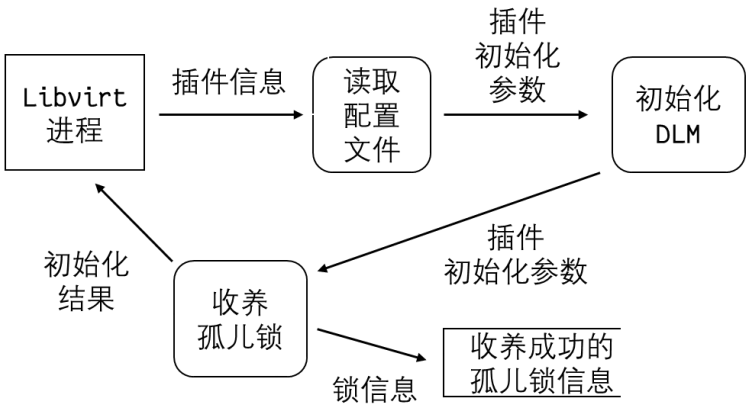


图 3-2 初始化数据流图

与初始化模块有关的数据字典如下所示：

名字：插件信息
描述：DLM-Corosync分布式锁管理插件的初始化信息
定义：插件信息 = 锁管理框架版本 + (配置文件地址) + (初始化参数)
位置：读取配置文件

名字：插件初始化参数
描述：DLM-Corosync插件的初始化参数
定义：插件初始化参数 = (自动加锁设置) + (锁空间名称)
+ (孤儿锁收养设置)
位置：初始化DLM、收养孤儿锁

名字：锁信息
描述：成功收养的孤儿锁信息
定义：锁信息 = 资源名称 + 资源进程id + 有效标志
位置：收养成功的孤儿锁信息

名字：初始化结果
描述：初始化模块操作的结果
定义：初始化模块 = [ 成功 | 失败 ]
位置：libvirt进程

### 3.2.2 资源关联模块

资源关联模块实现了锁与资源的相互关联。它利用需要加锁资源的信息，根据资源的种类产生一个独一无二的标识符，令特定的锁与之相互绑定，最后向 libvirtd 守护进程中返回这些产生的信息，这些信息用于在锁操作的上下文中使用。

资源关联模块的数据流图如下图 3-3 所示：

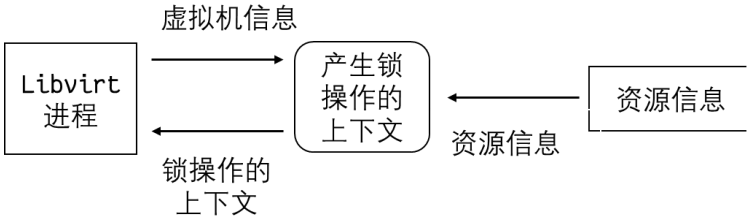


图 3-3 资源关联数据流图

与资源关联模块相关的数据字典如下所示：

名字：虚拟机信息
描述：资源所有者虚拟机的相关信息
定义：虚拟机信息 = uuid + id + name + pid + uri
位置：产生锁操作的上下文

名字：资源信息
描述：资源有关的信息
定义：资源信息 = uuid + id + name + pid + uri
位置：产生锁操作的上下文

名字：锁操作的上下文
描述：资源与锁关联后产生的信息，用于锁的操作模块
定义：锁操作的上下文 = 虚拟机信息 + 0{资源信息}
位置：产生锁操作的上下文

### 3.2.3 锁的操作模块

锁的操作模块主要用于负责分布式锁管理插件对外暴露出的 API 接口的具体实现，使得该插件在 libvirt 源码中更容易地被使用。获取操作是指利用资源管理模块获得的锁信息请求加锁，如果资源已经被加锁则返回失败标识符，否则返回成功标识符。释放操作也是利用资源管理模块获得的锁信息请求释放锁，如果与锁绑定的资源存在相应的锁，那么便会释放成功，否则失败。至于查询操作，该插件的并没有使用相关的功能，但为了框架的兼容性予以保留，因此它不执行任何具体的操作。

图 3-4 是锁的操作模块的数据流图：

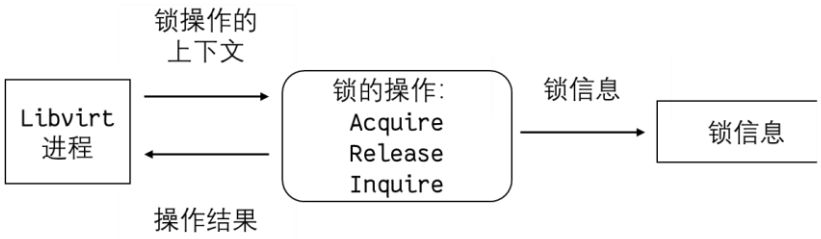


图 3-4 锁的操作数据流图

与锁的操作模块相关的数据字典如下所示：

名字：锁操作的上下文
描述：资源与锁关联后产生的信息，用于锁的操作模块
定义：锁操作的上下文 = 虚拟机信息 + 0{资源信息}
位置：锁的操作

| |
|---|
| 名字：锁信息<br>描述：成功收养的孤儿锁信息<br>定义：锁信息 = 资源名称 + 资源进程id + 有效标志<br>位置：锁信息 |

| |
|---|
| 名字：操作结果<br>描述：锁的操作结果<br>定义：操作结果 = [ 成功 \| 失败 ]<br>位置：libvirt进程 |

## 3.3 本章小结

本章通过分析 libvirt 分布式锁管理框架的应用场景以及需要达成的目标，对 DLM-Corosync 插件的功能需求进行了详细的分析，针对框架的每一点要求提出了可行的解决方案，并且使功能模块化，从而能够方便地设计初始化模块、资源关联模块以及锁的操作模块需要实现的功能。最后，通过数据字典和数据流图明晰各个子功能之间的逻辑关系，大致描述了该插件需要如何实现。

# 第四章　插件设计与实现

## 4.1　插件的总体设计

基于 DLM-Corosync 插件的需求分析和应用场景，以及 libvirt 项目的实际需要，插件以可插拔形式的动态链接库提供相应功能，相关功能的代码与 libvirt 的核心代码相互独立。

插件源码主要在 src/locking/lock_driver_dlm.c 文件中。该文件中的所有函数与变量拥有属于此文件内部的命名空间，通过对外暴露 virLockDriverImpl 来使用相关的功能。它使用自动化编译工具 autotools、automake 等系列软件进行自动化编译，最终生成动态链接库 dlm.so，由它提供相关的功能。

libvirt 可以通过在配置文件中配置 lock_manager 的参数为 dlm 来使用该插件。在 libvirt 的初始化过程中，若所管理的虚拟机监视器为 Xen 或 QEMU，则会读取相应的配置文件，对 Xen 来说是 libxl.conf，对 QEMU 来说是 qemu.conf，判断其中的 lock_manager 参数是否为 dlm，若是则进行 dlm 插件的初始化操作，否则根据其配置参数进行相应的插件初始化工作。Dlm 插件的初始化流程图如图 4-1：

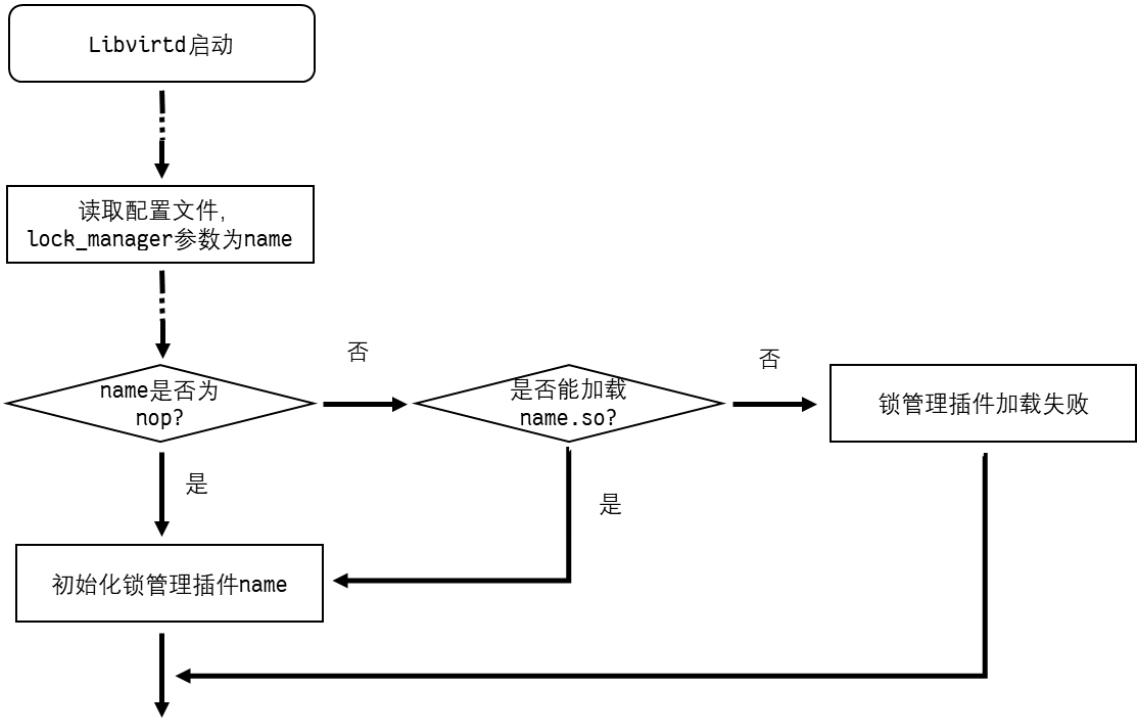


图 4-1 锁管理插件初始化流程图

## 4.2 插件数据结构的设计与实现

为了便于组织锁与锁资源关联的有关信息，设计了以下的数据结构用于存储相应的数据集：

- virLockManagerDLMLock
- virLockManagerDLMLockResource
- virLockManagerDLMResource
- virLockManagerDLMPrivate
- virLockManagerDLMDriver

他们的含义如下表 4-1、表 4-2、表 4-3、表 4-4 与表 4-5 中的描述所示：

表 4-1 virLockManagerDLMLock 结构体信息

| **用途** | 存储锁的详细信息 | |
|---|---|---|
| **成员** | **数据类型** | **含义** |
| vm_pid | pid_t | 与锁关联的资源所有者进程标识 |
| unsigned int | unsigned int | 锁的标识符 |

表 4-2 virLockManagerDLMLockResource 结构体信息

| **用途** | 存储资源与锁相关联的有关信息 | |
|---|---|---|
| **成员** | **数据类型** | **含义** |
| name | char * | 资源名称 |
| mode | unsigned int | 资源关联的锁类型 |
| nHolders | size_t | 读写类型锁的锁数量 |
| nLocks | size_t | 所有类型锁的数量 |
| locks | virLockManagerDLMLockPtr | 指向存储锁的详细信息区域 |

表 4-3 virLockManagerDLMResource 结构体信息

| **用途** | 用于在锁操作的上下文中临时存储与锁操作有关的信息 | |
|---|---|---|
| **成员** | **数据类型** | **含义** |
| name | char * | 资源名称 |
| Mode | unsigned int | 锁的模式 |

表 4-4 virLockManagerDLMPrivate 结构体信息

| 用途 | 在锁操作的上下文中存储与资源所有者的有关信息 | |
|---|---|---|
| **成员** | **数据类型** | **含义** |
| vm_uuid | unsigned char | 资源所有者的 uuid |
| vm_name | char * | 资源所有者的名称 |
| vm_pid | pid_t | 资源所有者进程的 pid |
| vm_id | int | 资源所有者的 id |
| nresources | size_t | 资源的数量 |
| resources | virLockManagerDLMResourcesPtr | 资源的信息 |
| hasRWDisks | bool | 访问权限标志 |

表 4-5 virLockManagerDLMDriver 结构体信息

| 用途 | 存储插件的参数信息以及所管理器的一些信息 | |
|---|---|---|
| **成员** | **数据类型** | **含义** |
| autoDiskLease | bool | 是否启用自动加锁 |
| requireLeaseForDisks | bool | 是否启用手动加锁 |
| purgeLockspace | bool | 是否在收养孤儿锁后释放未收养的孤儿锁 |
| lockspaceName | char * | DLM 锁空间名称 |
| lockspace | dlm_lshandle_t | 用于 DLM 锁操作的参数 |
| resources | virHashTablePtr | 用于存储锁信息的哈希表指针 |
| lockFd | int | 锁操作记录文件的文件描述符 |

## 4.3 初始化模块的设计与实现

初始化模块对外暴露出两个接口：drvInit 与 drvDeinit。

他们的函数原型与作用如表 4-6 所示：

表 4-6 初始化模块中 drvInit 与 drvDeinit 的函数定义

| 接口名称 | 函数原型 | 作用 |
|---|---|---|
| drvInit | int (*virLockDriverInit)(unsigned int version,<br>const char *configFile,<br>unsigned int flags); | 初始化插件 |
| drvDeinit | int (*virLockDriverDeinit)(void); | 卸载插件 |

drvInit 为函数 virLockManagerDLMInit 的函数指针，用以提供初始化功能。如果初始化成功则返回 0，失败则返回-1。

它的函数参数及其相关含义如表 4-7 所示：

表 4-7 virLockManagerDLMInit 函数的参数

| 参数名称 | 类型 | 含义 |
|---|---|---|
| version | unsigned int | Libvirt 请求的插件版本 |
| configFile | const char * | Libvirt 请求的插件配置文件地址 |
| Flags | unsigned int | Libvirt 请求的插件参数 |

drvDeinit 为函数 virLockManagerDLMDeinit 的函数指针，用以提供卸载功能，主要用于清理初始化之后以及运行过程中动态分配的内存空间。函数返回值为 void，并且相应的函数参数为 NULL。

在初始化模块中，相应的流程为：首先检查是否存在 driver 变量，此变量是否为 NULL 代表着 libvirt 分布式锁插件模块是否已经初始化。若 driver 为 NULL，则代表插件未经初始化，需要进行如下描述的插件初始化操作——首先读取 configFile 字符串变量指向的配置文件，如果能够成功打开该文件，则读取记录在文件中的相应初始化参数，否则使用默认的配置参数，默认的配置参数硬编码在源代码中。在此之后，需要进行 DLM 插件的初始化工作。DLM 的初始化工作主要包括三部分：创建用于存储锁信息的哈希表，打开 DLM 锁空间和设置 DLM 守护进程 dlm_controld 的回调通知线程，收养孤儿锁与设置锁信息记录文件。在以上的初始化过程中，任何一步如果产生了不可恢复的错误，则必须进行卸载操作，并且返回初始化失败标识符。

详细的初始化流程图如图 4-2 所示:

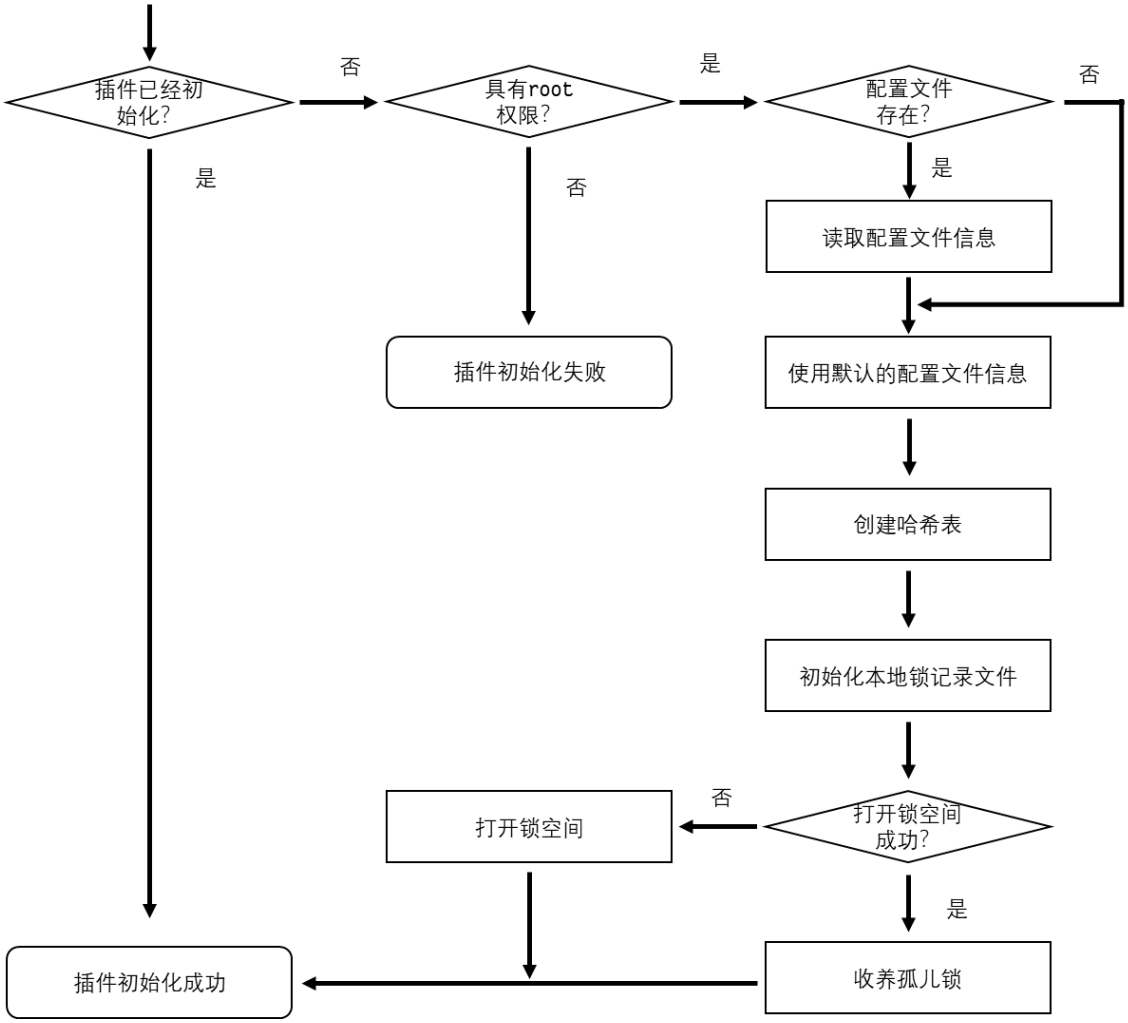


图 4-2 锁插件初始化流程图

相应的卸载的流程为：首先确保 driver 变量非 NULL。若为 NULL 则表明插件未曾初始化，插件卸载成功。否则依次判断锁空间是否打开、记录锁信息的哈希表是否存在、本地锁信息记录文件是否打开，根据判断结果决定是否需要关闭 DLM 锁空间与通知 DLM 回调通知线程退出，是否需要释放存储锁信息的哈希表，以及是否需要关闭记录锁信息文件的文件描述符，之后释放为 driver 变量申请的动态内存。经过上述的一系列操作流程，确保了卸载插件之后不会有内存泄漏的情况发生。

相应的流程图如图 4-3 所示：

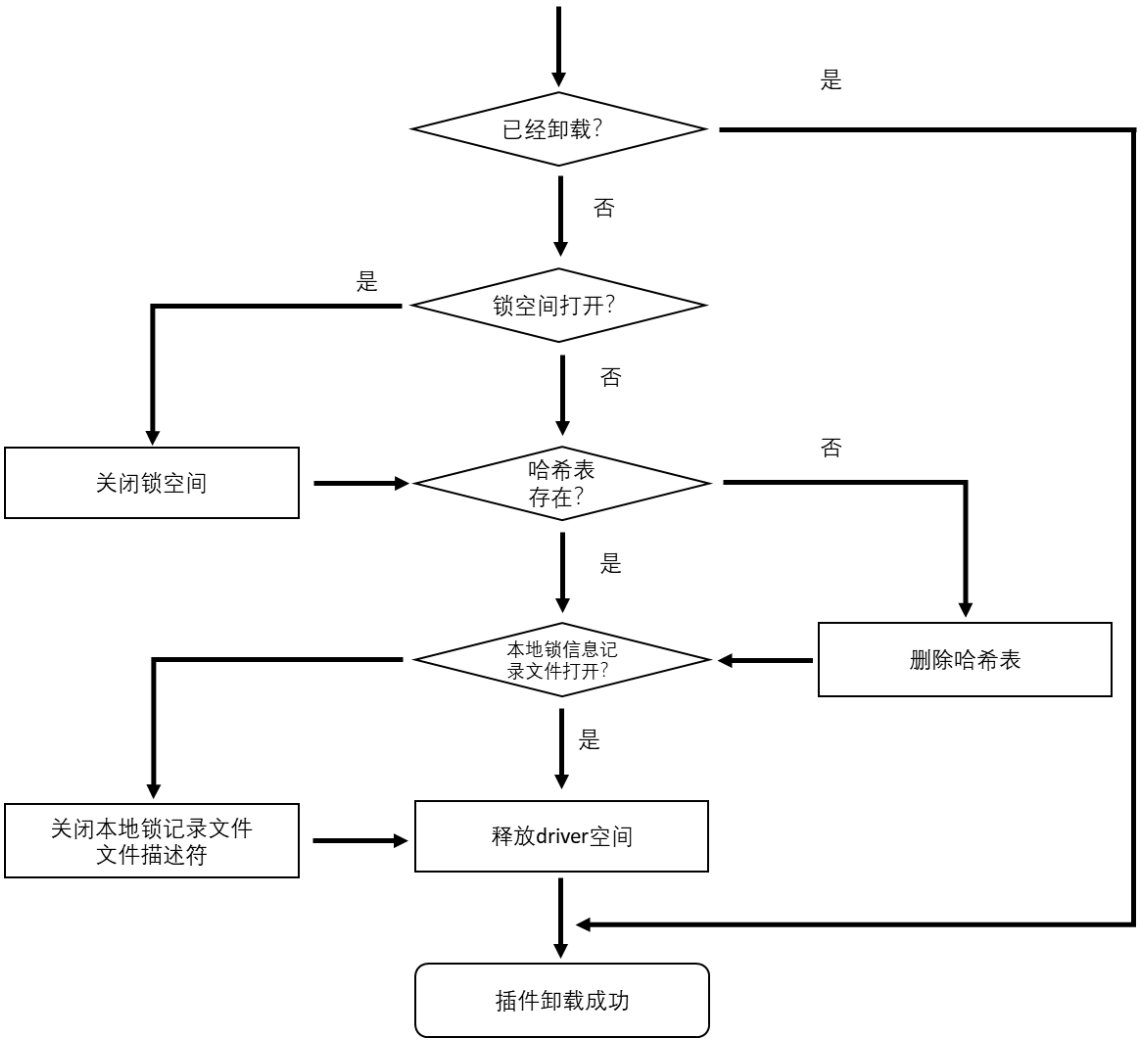


图 4-3 锁插件卸载流程图

## 4.4 资源关联模块的设计与实现

资源关联模块对外暴露出三个接口：drvNew、drvFree 与 drvAddResource。他们的函数原型与作用如表 4-8 所示：

表 4-8 资源关联模块中 drvNew、drvFree 与 drvAddResource 的函数定义

| 接口名称 | 函数原型 | 作用 |
|---|---|---|
| drvNew | int (*virLockDriverNew)(<br>virLockManagerPtr lock,<br>unsigned int type,<br>size_t nparams,<br>virLockManagerParamPtr params,<br>unsigned int flags); | 初始化锁操作的进程上下文 |

| | | |
|---|---|---|
| drvFree | void (*virLockDriverFree)(virLockManagerPtr lock); | 删除锁操作的进程上下文 |
| drvAddResource | int (*virLockDriverAddResource)(<br>virLockManagerPtr lock,<br>unsigned int type,<br>const char *name,<br>size_t nparams,<br>virLockManagerParamPtr params,<br>unsigned int flags); | 将锁资源与锁信息的相互关联，在锁的操作之前使用 |

其中，drvNew 为函数 virLockManagerDLMNew 的函数指针，它用于初始化对锁操作的进程上下文，如果初始化成功则返回 0，失败则返回-1。

它的函数参数如表 4-9 所示：

表 4-9 virLockManagerDLMNew 函数的参数

| 参数名称 | 类型 | 含义 |
|---|---|---|
| lock | virLockManagerPtr | 锁的操作的上下文数据区域指针 |
| type | unsigned int | 资源所有者的进程类型标志 |
| nparams | size_t | 资源所有者元信息参数的数量 |
| params | virLockManagerParamPtr | 资源所有者的元信息参数数组指针 |
| flags | unsigned int | 与 drvNew 有关的请求参数标志 |

与其有关的具体流程为：首先检查插件是否已经初始化，即 driver 是否为 NULL，如果不为 NULL 便令 privateData 指向一块可以使用的内存区域，以便存储资源与锁的关联信息，用于接下来的锁的操作函数使用。在该函数中仅拷贝部分信息，都与虚拟机进程实例有关，他们分别是：uuid、name、id、pid 和 uri，之后检测这些信息是否有效。在此过程中，无论 driver 为 NULL 还是检测信息的过程中发现了无效信息，都代表初始化锁操作的进程上下文失败。

与此相关的流程图如图 4-4 所示：

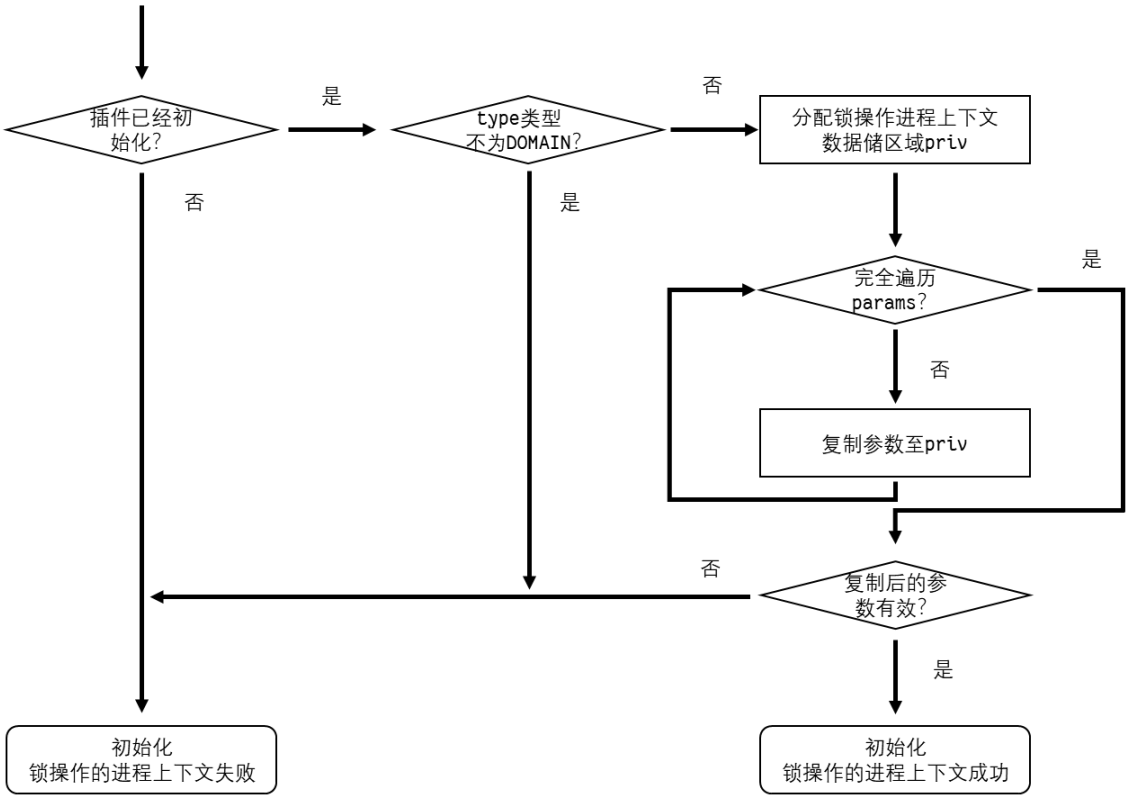


图 4-4 初始化锁操作的进程上下文流程图

drvFree 为函数 virLockManagerDLMFree 的函数指针，它用于删除对锁操作的进程上下文，返回值为 void，即不存在返回值。

它的函数参数如表 4-10 所示：

表 4-10 virLockManagerDLMFree 函数的参数

| 参数名称 | 类型 | 含义 |
|---|---|---|
| lock | virLockManagerPtr | 锁操作的上下文数据区域指针 |

与其有关的具体流程为：首先通过结构体 lock 的成员 privateData 获得锁操作的上下文的数据区域指针 priv，进而判断 priv 是否有效，如果有效则依次释放锁信息存储区域、虚拟机名称字符串空间和 priv 本身。

相关的流程图如图 4-5 所示：

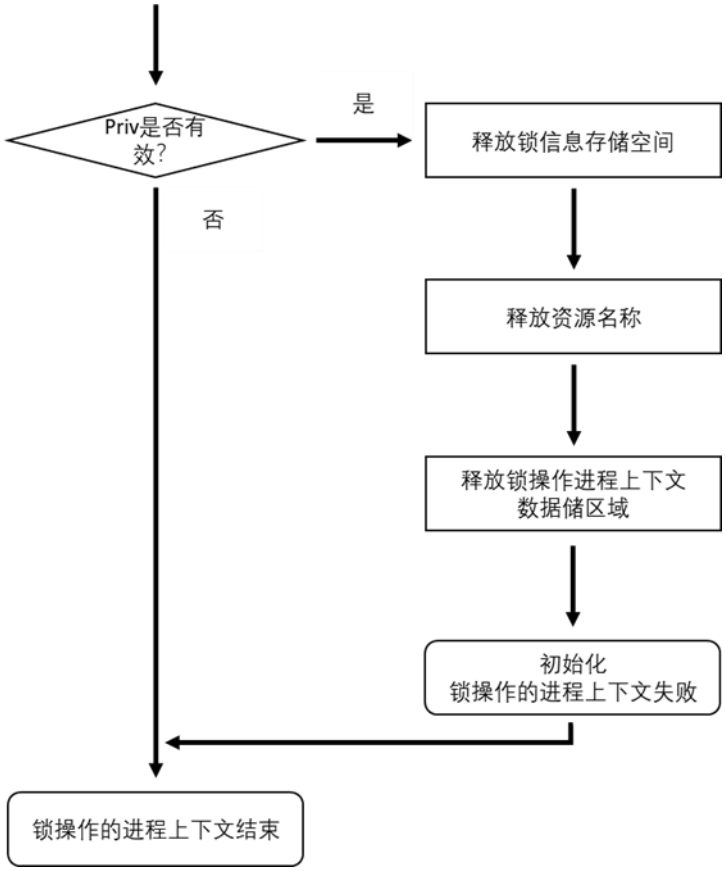


图 4-5 删除锁操作的上下文流程图

drvAddResource 为函数 virLockManagerDLMAddResource 的函数指针，它用于将锁资源对象与锁信息相互关联。如果函数调用成功则返回 0，失败的情况下返回 -1。

它的函数参数如表 4-11 所示：

表 4-11 virLockManagerDLMAddResource 的函数参数

| 参数名称 | 类型 | 含义 |
|---|---|---|
| lock | virLockManagerPtr | 锁操作的上下文数据区域指针 |
| type | unsigned int | 资源的种类 |
| name | const char | 资源名称 |
| nparams | size_t | 资源的元信息数量 |
| params | virLockManagerParamPtr | 资源的元信息参数 |
| flags | unsigned int | 资源的访问权限标志 |

virLockManagerDLMAddResource 将资源信息与锁相关联，并存储在锁操作的上下文数据存储区域中的指针 resources 指向的内存区域中。与其有关的具体流程

为：首先判断资源访问权限是否为只读，如果是则直接返回调用成功标志，否则会根据资源的种类决定下一步的具体操作。若资源类型为 RESOURCE_TYPE_DISK，接下来判断插件是否设置了自动加锁，如果是则根据资源名称生成独一无二的哈希值，否则在屏幕上输出有关的警告信息，并返回调用失败标识符；若资源的类型为 RESOURCE_TYPE_LEASE，则直接拷贝资源名称。之后再根据资源是否共享为下文中用到的 DLM 锁信息设置相应的模式类型，最后把资源的名称与模式信息存储在 resources 指向的内存区域中，即关联锁信息。

相关的流程图如图 4-6 所示：

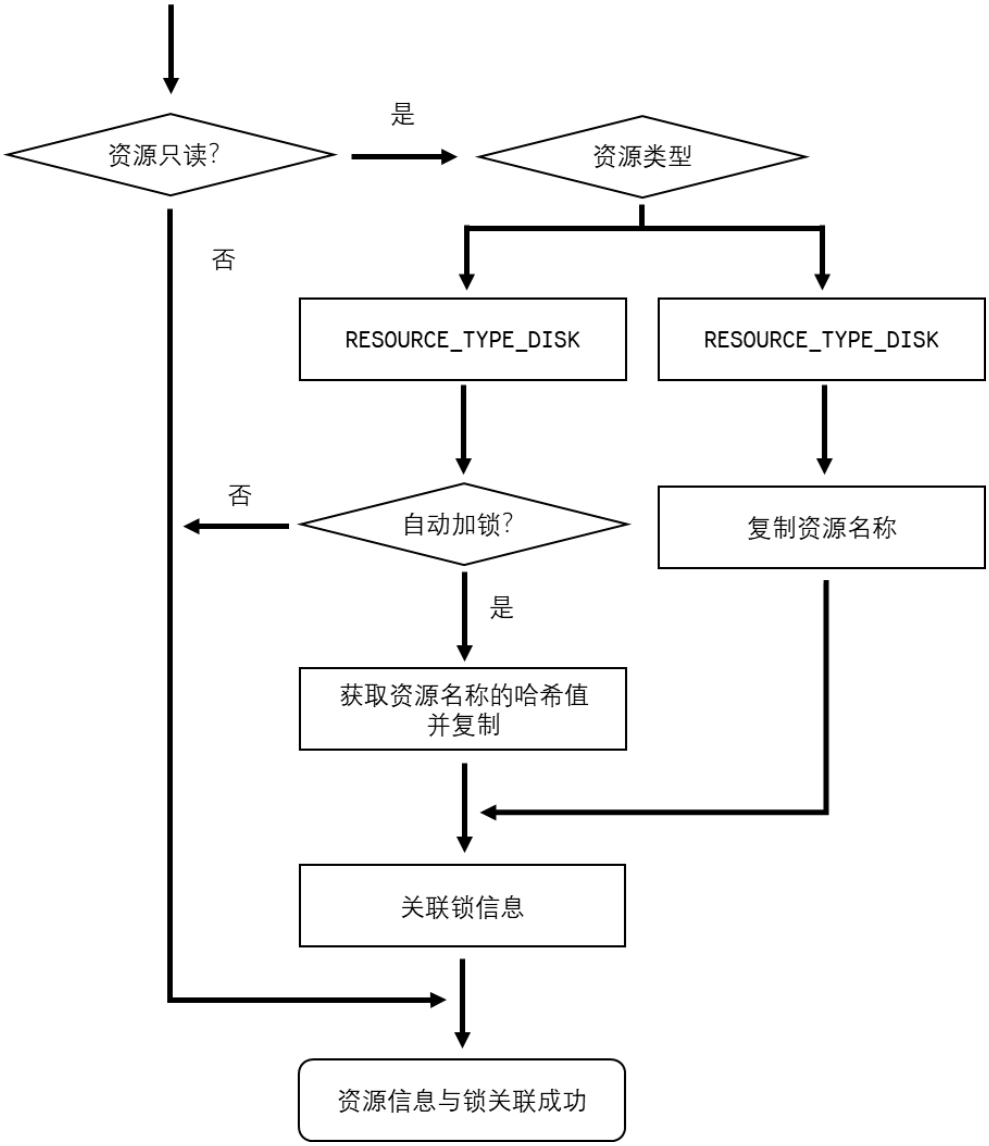


图 4-6 资源信息与锁相关联流程图

## 4.5 锁的操作模块的设计与实现

锁的操作模块对外暴露出三个接口：drvAcquire、drvRelease 和 drvInquire。他们的函数原型与作用如表 4-12 所示：

表 4-12 锁的操作模块中 drvAcquire、drvRelease 和 drvInquire 的函数定义

| 接口名称 | 函数原型 | 作用 |
|---|---|---|
| drvAcquire | int (*virLockDriverAcquire)(<br>virLockManagerPtr lock,<br>const char *state,<br>unsigned int flags,<br>virDomainLockFailureAction action,<br>int *fd); | 向锁管理器申请加锁 |
| drvRelease | int (*virLockDriverRelease)(<br>virLockManagerPtr lock,<br>char **state,<br>unsigned int flags); | 向锁管理器申请释放锁 |
| drvInquire | int (*virLockDriverInquire)(<br>virLockManagerPtr lock,<br>char **state,<br>unsigned int flags); | 向锁管理器询问锁状态 |

drvAcquire 为 virLockManagerDLMAcquire 的函数指针，用于向锁管理器申请加锁，如果成功返回 0，失败则返回-1。

它的函数参数如表 4-13 所示：

表 4-13 virLockManagerDLMAcquire 的函数参数

| 参数名称 | 类型 | 含义 |
|---|---|---|
| lock | virLockManagerPtr | 锁操作的上下文数据区域指针 |
| state | const char * | 当前的锁状态 |
| flags | unsigned int | 可选的标志 |
| action | virDomainLockFailureAction | 当锁失联时要执行的动作 |
| fd | int * | 可选，需要返回的文件描述符 |

与该函数有关的详细流程为：如果设置了手动加锁，并且设置了资源可读写，

却无相应的锁信息，此时会报错。之后再判断 DLM 锁空间是否打开，如果则根据不同的 flags 标志执行不同的动作。如果 Flags 为 ACQUIRE_RESTRICT，说明该进程为 libvirtd 的子进程，在接下来的过程中会通过类似于 exec 的系统调用变为虚拟机进程继续运行，因此在此时要做的是关闭 DLM 锁空间；若不为 ACQUIRE_REGISTER_ONLY，说明需要执行申请加锁操作，此时通过首先遍历 priv->resources 中的锁信息，判断哈希表中是否存在相应的锁信息条目，如果存在不为 NL 类型的锁则需要创建并记录一个 NL 锁条目，否则继续下一步操作——取得 NL 类型的锁信息，再将其转换为锁请求的锁模式。

相关的流程图如图 4-7 所示：

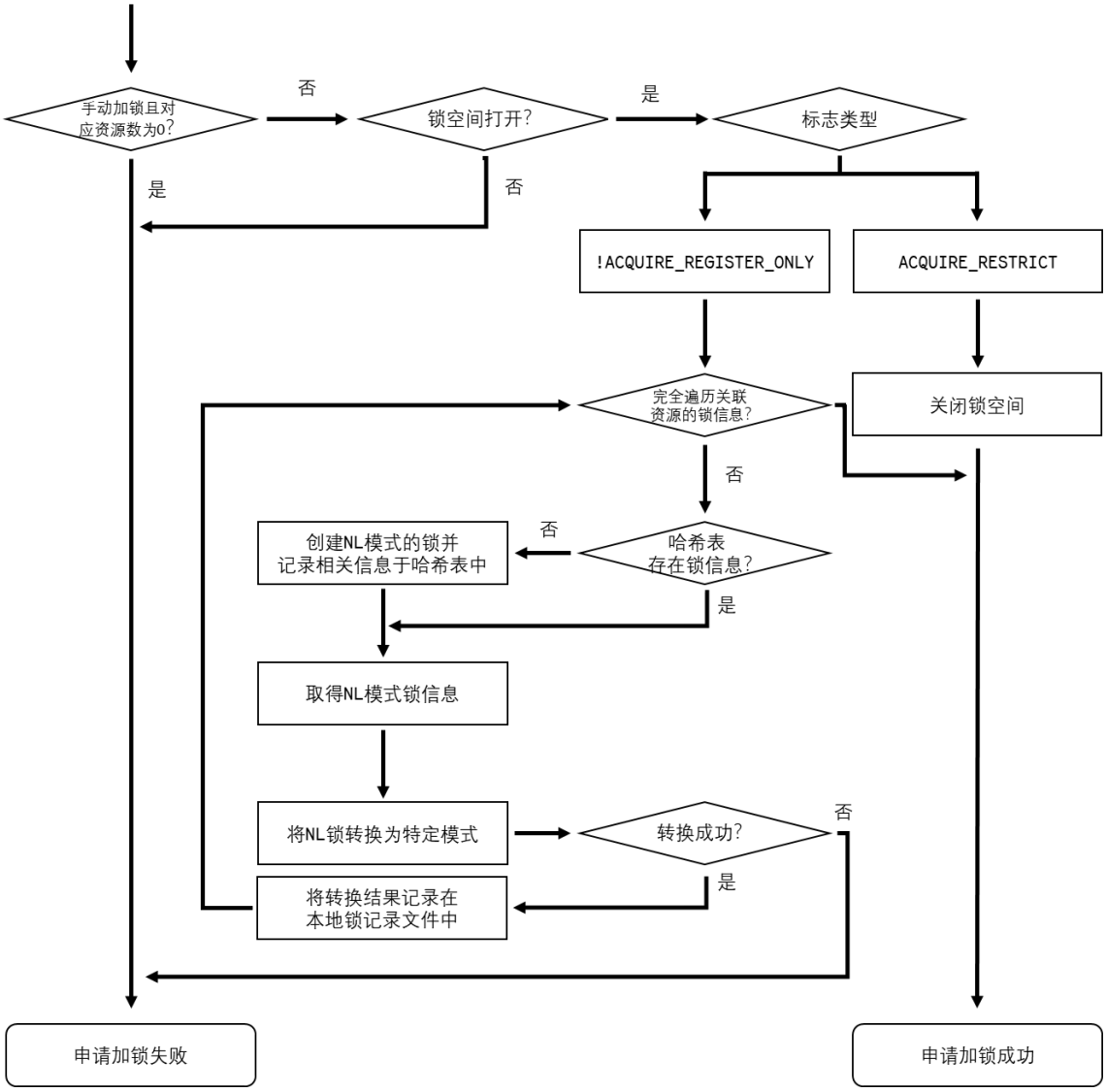


图 4-7 申请加锁流程图

drvRelease 为 virLockManagerDLMRelease 的函数指针，用于向锁管理器申请释放锁，如果执行成功返回 0，失败返回-1。

它的函数参数如表 4-14 所示：

表 4-14 virLockManagerDLMRelease 的函数参数

| 参数名称 | 类型 | 含义 |
|---|---|---|
| lock | virLockManagerPtr | 锁操作的上下文数据区域指针。 |
| state | char ** | 锁状态的字符串指针。 |
| flags | unsigned int | 可选的标志。 |

与其有关的详细流程为：先检查 DLM 锁空间是否已经打开，如果打开则遍历 priv->resources 中的锁信息，检查哈希表中是否存在相应的锁信息。如果存在该锁信息则请求 DLM 锁管理器释放该锁，成功移除之后再从哈希表中移除相关信息，然后继续下一个锁直至完全遍历结束。相关流程图如图 4-8 所示：

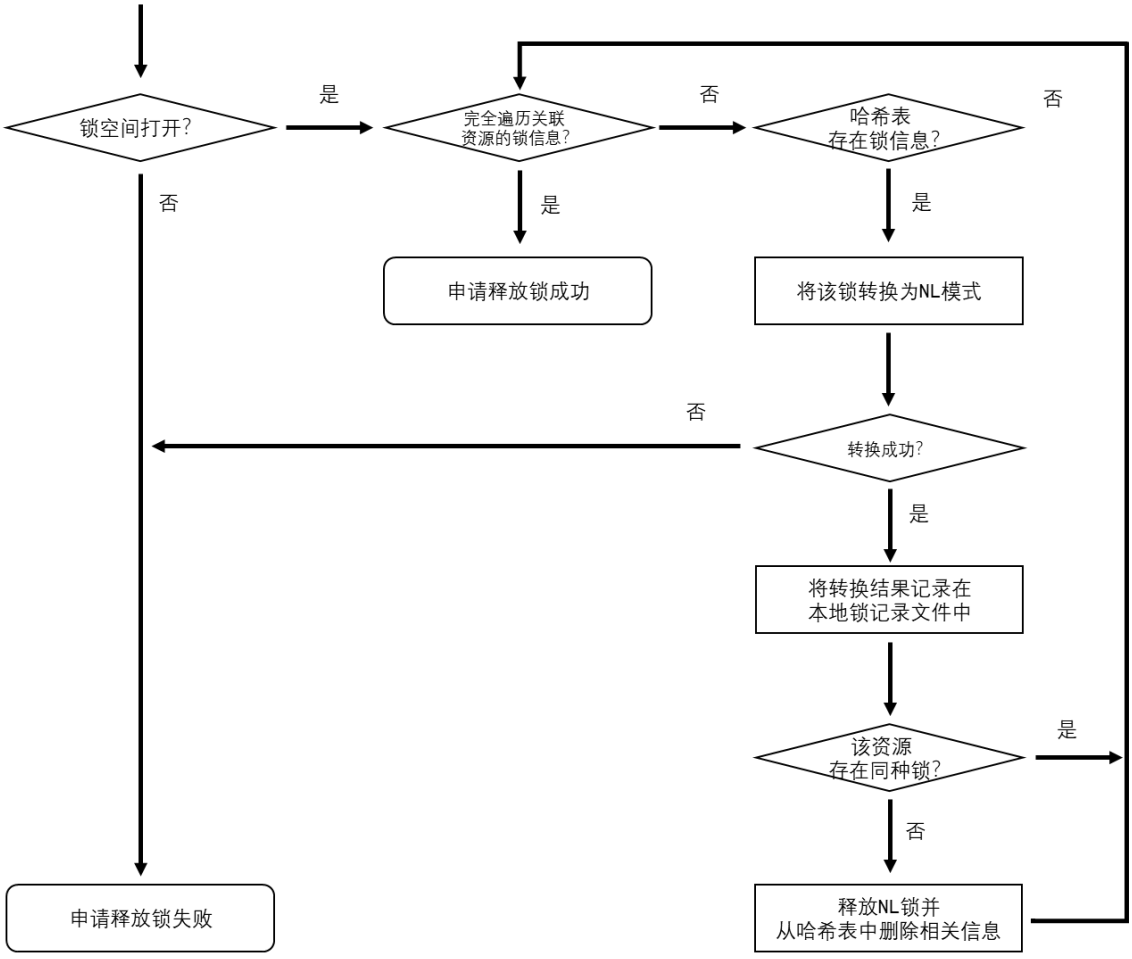


图 4-8 申请释放锁流程图

drvInquire 为 virLockManagerDLMInquire 的函数指针，它用于向锁管理器询问锁状态，成功返回 0，失败返回-1。

它的函数参数如表 4-9 所示：

表 4-9 virLockManagerDLMInquire 的函数参数

| 参数名称 | 类型 | 含义 |
| --- | --- | --- |
| lock | virLockManagerPtr | 锁操作的上下文数据区域指针。 |
| state | char ** | 锁状态的字符串指针。 |
| flags | unsigned int | 可选的标志，当前并未使用。 |

virLockManagerDLMInquire 函数在 DLM-Corosync 插件中并不起任何作用，仅仅将 state 赋值为 NULL，但为了兼容性该函数予以保留。

## 4.6 本章小结

本章主要介绍了插件各个模块的详细设计与实现。通过对插件总体结构的分析，将插件实现分为插件的数据结构、初始化模块、资源关联模块与锁的操作模块四个部分分别阐述详细的设计，并采用流程图的方式详细描述各个功能模块的实现。

# 第五章　插件的测试

## 5.1 测试目标

插件功能测试是对本插件各功能模块的逻辑功能进行测试。本测试采用黑盒测试的方法，不考虑插件的内部实现，只检查插件是否能正确地接受操作指令并产生正确的结果。

在 libvirt 虚拟机的生命周期中，从一个状态切换到另一个状态的具体实现上，可能会存在有关锁操作的代码。Libvirt 管理虚拟机的命令主要是 start、create、shutdown、destroy、suspend、resume、save、restore、migrate 等几大类。由于每个类别都含有很多参数，不可能一一进行测试，因此只测试该插件是否能达到实现资源互斥的目标。

## 5.2 测试用例

### 5.2.1 测试环境

集群中存在两个节点，每个节点配置如下：Debian Testing 操作系统（最后更新日期：2018 年 5 月 9 日），libvirt 版本为 4.2.0（已打上 Corosync-DLM 锁插件的功能补丁），QEMU 版本为 2.8.1，DLM 版本 4.0.7，Corosync 版本为 2.4.4，virt-manager 版本为 1.4.3。集群节点间的关系如下图 5-1 所示：

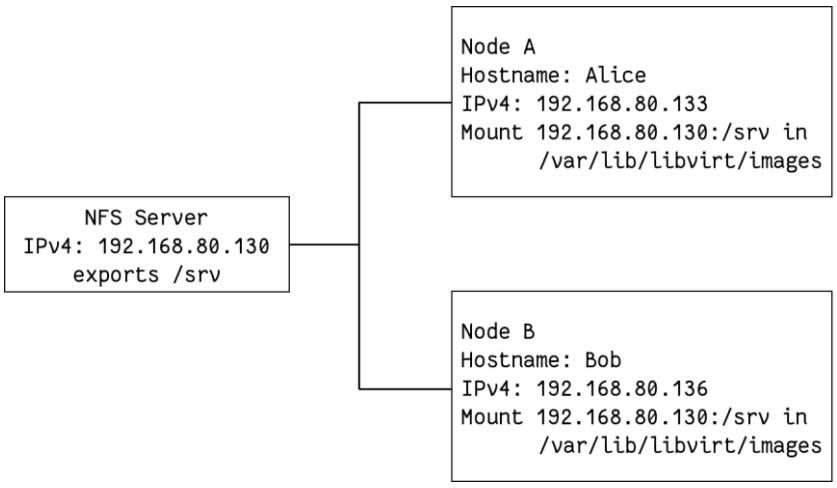


图 5-1 测试环境

### 5.2.2 同一节点上的测试

此测试只在 Node A 上进行，创建了除了所使用的磁盘镜像之外其余配置完全相同的 4 个虚拟机，分别为 vm-1、vm-2、vm-3、vm-4。磁盘镜像有三个，文件名称分别为 one.qcow2、two.qcow2、share.raw。用于测试是否可以达到以下目标：

- 在同一节点中，是否可以阻止非共享磁盘镜像在不同虚拟机上使用。
- 在同一节点中，是否允许共享磁盘镜像在不同虚拟机上使用。

下表 5-1 是该黑盒测试的详细测试结果：

表 5-1 同一节点上的测试用例

| 用例编号 | 用例说明 | 预期结果 | 实际结果 |
|---|---|---|---|
| vm-1 | 使用可读写的 one.qcow2 与 share.raw，其中 share.raw 设置为共享 | vm-1 启动成功 | Domain vm-1 started |
| vm-2 | 在 vm-1 启动成功的前提下，使用可读写的 one.qcow2 | vm-2 启动失败 | error: Failed to start domain vm-2<br>error: internal error: failed to acquire lock: the lock could not be granted |
| vm-3 | 在 vm-1 启动成功的前提下，使用只读的 one.qcow2 | vm-3 启动成功 | Domain vm-3 started |
| vm-4 | 在 vm-1 启动成功的前提下，使用可读写的 share.raw，share.raw 设置为共享 | vm-4 启动成功 | Domain vm-4 started |

图 5-1 是相应的测试结果图，如下所示：

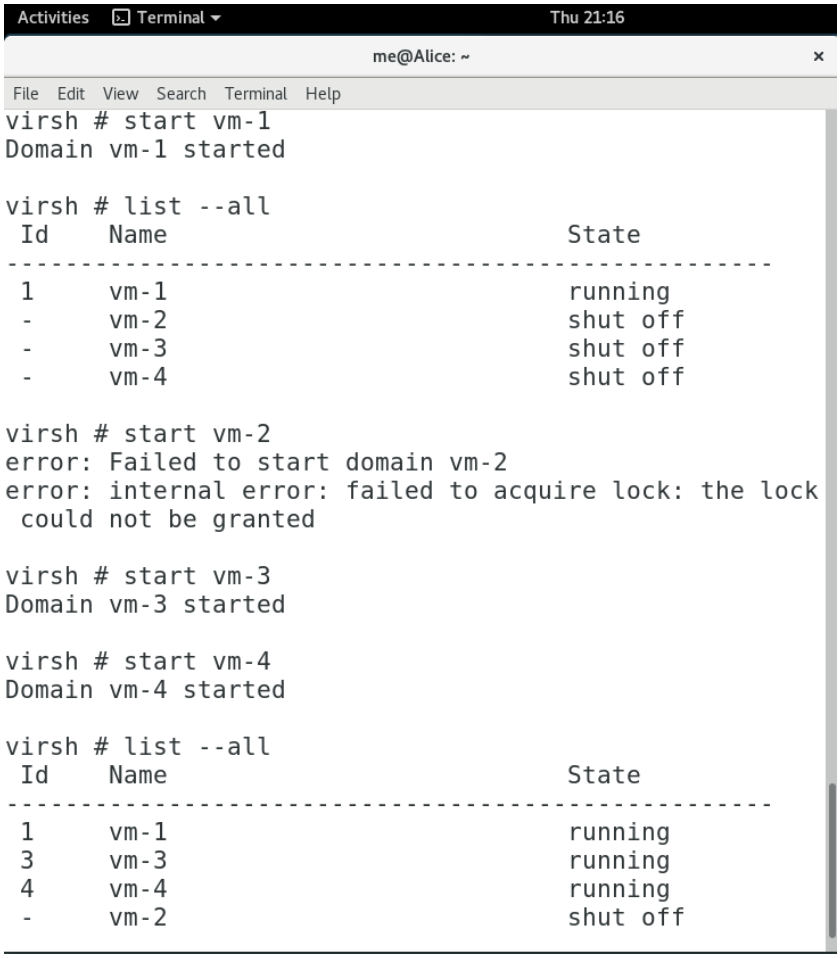


图 5-1 同一节点上的测试用例结果

### 5.2.3 不同节点上的测试

此测试在 Node A 和 Node B 上组成的集群上进行，创建了除所使用的磁盘镜像之外其余配置完全相同的 4 个虚拟机，分别为 vm-1、vm-2、vm-3、vm-4。其中，通过 live migration 将正在运行的虚拟机 vm-1 迁移到 Node B 上，之后在 Node A 启动 vm-2、vm-3、vm-4。磁盘镜像有三个，文件名称分别为 one.qcow2、two.qcow2、share.raw，通过 NFS Server 在节点中共享使用。用于测试是否可以达到以下目标：

- 是否可以阻止同一个虚拟机在不同节点上启动。
- 是否可以阻止非共享磁盘镜像在不同节点上且不同的虚拟机上使用。
- 是否允许共享磁盘镜像在不同节点上且不同的虚拟机上使用。

下表 5-2 是详细的测试结果：

表 5-2 不同节点上的测试用例

| 用例编号 | 用例说明 | 预期结果 | 实际结果 |
|---|---|---|---|
| vm-1-B | 定义在 Node A 上，使用可读写的 one.qcow2 与 share.raw，其中 share.raw 设置为共享，启动后迁移到 Node B 之上 | vm-1-B 启动并迁移成功 | Domain vm-1 started<br>迁移成功 |
| vm-1-A | 与 vim-1B 有着相同的定义 | vm-1-A 启动失败 | error: Failed to start domain vm-1<br>error: internal error: failed to acquire lock: the lock could not be granted |
| vm-2 | 在 vm-1 启动成功的前提下，使用可读写的 one.qcow2 | vm-2 启动失败 | error: Failed to start domain vm-2<br>error: internal error: failed to acquire lock: the lock could not be granted |
| vm-3 | 在 vm-1 启动成功的前提下，使用只读的 one.qcow2 | vm-3 启动成功 | Domain vm-3 started |
| vm-4 | 在 vm-1 启动成功的前提下，使用可读写的 share.raw，share.raw 设置为共享 | vm-4 启动成功 | Domain vm-4 started |

图 5-2 中是相应的测试结果图，如下所示：

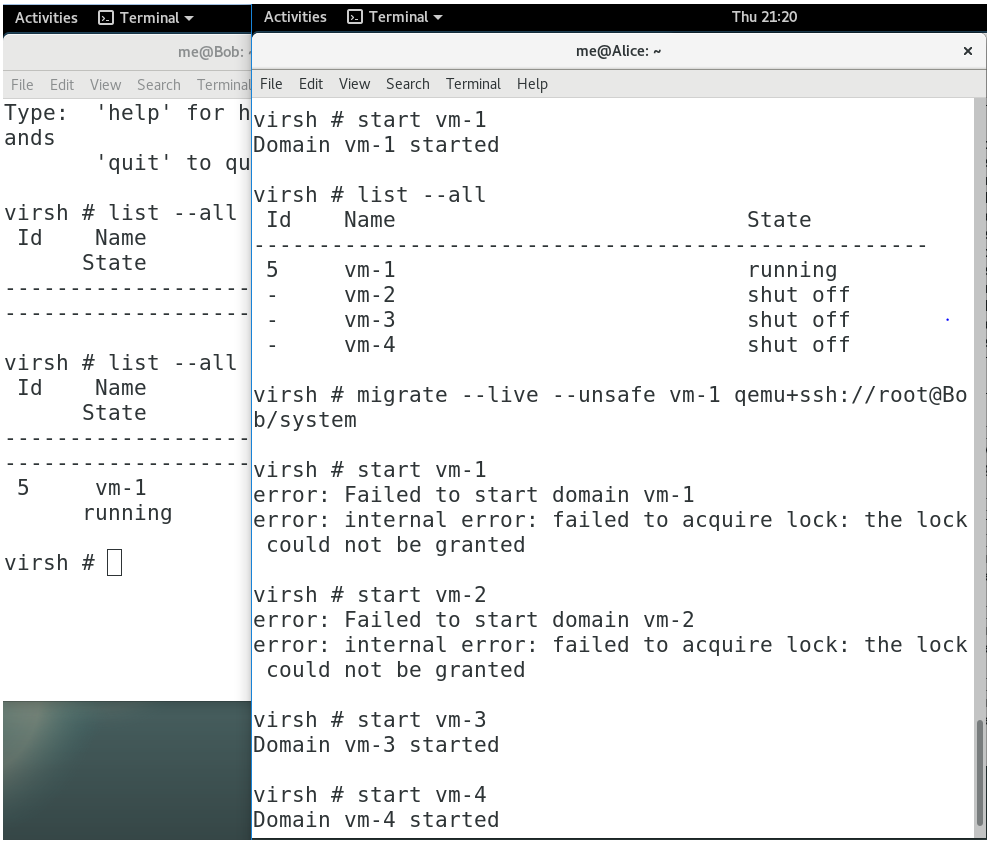


图 5-2 不同节点上的测试结果用例

## 5.3 本章小结

本章主要对系统进行功能测试。功能测试围绕插件需要完成的目标进行黑盒测试，用以检测实现了的插件是否能够正确地接收操作指令并产生正确的结果。

# 第六章 总结与展望

## 6.1 论文总结

本论文通过前五个章节先后分析了 libvirt 的分布式管理锁插件 DLM-Corosync 的实现，完成了插件的功能设计并整合进 libvirt 源码中，可以通过在 libvirt 相应的配置文件中启用该插件。

插件的开发是在 Debian 9 这个 Linux 发行版下进行的，编程语言是 C 语言，libvirt 版本为 4.1.0，采用自动化编译工具 autotools、automake 等工具整合进 libvirt 源码中，以 git patch 的形式发布源码。论文主要完成的工作总结如下：

1. 介绍了项目的研究背景和意义。对虚拟化管理工具 libvirt 及其锁管理框架的国内外研究现状做了简要介绍与分析，并阐明本论文的具体内容展开和主要工作。

2. 介绍了项目涉及到的系统关键技术。介绍了项目所用到的系统关键技术：高可用与虚拟化，概述了 libvirt、DLM 与 Corosync 的一些技术细节与 API 使用说明。

3. 完成了项目的需求分析与设计。通过分析项目的应用场景以及需要达成的目标，把功能模块分为初始化模块、资源关联模块与锁操作模块三部分，使用数据流图和数据字典描述各模块的总体设计方案。

4. 阐述了项目的详细设计与实现。首先通过表格对插件的数据结构与各模块的参数进行了简单介绍，再采用流程图并辅以简单的文字描述的方式说明了各个模块的内部工作流程。

5. 对实现的插件进行了黑盒测试。在这部分首先描述了测试的目标，再在单机与分布式两种环境下采用黑盒测试的方法检验插件是否能正确地接受操作指令并产生正确的结果。

## 6.2 展望

尽管 libvirt 设计了锁机制用于保证资源的互斥，但仍然会出现不同的进程对

同一个资源写入的情况，比如使用 qemu-img，这种情况导致了不少 bug 报告的产生。因此有人希望 libvirt 设计一种机制，以便其他进程能够从 libvirt 中获取到资源加锁的有关信息。不幸的是，上游认为这个功能不属于 libvirt 的职责范围，给与了否决[23]。

自 2016 年 4 月开始，有人给 QEMU 提出了一系列名称为 block: Lock images when opening 的补丁[24]，为的是在 QEMU 中添加上述功能以解决资源互斥的情况。这项功能经过若干版本的迭代，最终在 2017 年 5 月进入主线代码。因此在某种程度上来说，2.10 版本后的 QEMU 与 libvirt 的锁机制存在的功能冲突。

本论文利用 Corosync 与 DLM 给出了 libvirt 锁管理框架的另一种实现，但在系统设计与实现上还存在一定的问题，需要进一步的讨论与完善。要继续的工作如下：

1. 此实现对 2.10 版本以后的 QEMU 存在兼容性问题，不能很好地与其协调工作，这点需要进一步地改善。

2. Libvirt 的锁管理框架适用于 QEMU 与 Xen，当前只做了 QEMU 下的测试，在 Xen 下是否同样有效还不明确，需要继续调研。

3. 对于实现了资源互斥版本的 QEMU，libvirt 锁管理框架是否还有存在的意义，有待进一步地考察。

4. 2.10 版本后的 QEMU 在不依赖 libvirt 的情况下能够保证资源的互斥。那么在 Xen 中，目前是否存在同样的特性？假如不存在，能否为 Xen 实现相同的功能特性呢？这些问题也需要做进一步调查。

# 致 谢

本次毕业设计的课题来源于我的实习公司 SUSE(Beijing)，由衷地感谢Roger(周志强)对我的选题帮助，任振、何刚、马林等人在我碰到困难时对我的悉心指导。

也很感谢王老师在校内对我整个毕业设计期间工作的细心指导，在我遇到问题时给予了无私的帮助，在本论文的撰写上也严格把关，逻辑严谨，条理清晰，循循善诱，印象十分深刻。

同时感谢在毕业设计期间和我共同学习，帮助我、给予我支持的同学们，衷心地感谢一直帮助我的陈瑶、王维昊、王文硕、石光银，还有室友刘铂、王嘉明，也有欧作松、高向珊，等等等等。更感谢一直关心我的家人与朋友。

最后感谢在百忙之中评阅论文的各位老师！

# 参考文献

[1] 杨望仙，朱定局，谢毅等，虚拟化技术在云计算中的研究进展，先进技术研究通报，2010 年 10 月，第 8 期

[2] M. Jones，Libvirt 虚拟化库剖析——针对简单的 Linux 虚拟化的 API，2010 年 02 月，https://www.ibm.com/developerworks/cn/linux/l-libvirt/index.html

[3] Aaron Chen，Svito，SherlockHolo 等，Libvirt (简体中文)，ArchWiki，2018 年 01 月，https://wiki.archlinux.org/index.php/Libvirt_(简体中文)

[4] Libvirt Project，Virtual machine lock manager，libvirt Docs，2018 年 7 月，https://libvirt.org/locking.html

[5] David Teigland，[libvirt] using sync_manager with libvirt，2010 年 08 月，https://www.redhat.com/archives/libvir-list/2010-August/msg00179.html

[6] Daniel P. Berrange，[libvirt] RFC: A proposal for lock manager plugins，2010 年 09 月，https://www.redhat.com/archives/libvir-list/2010-September/msg00167.html

[7] Daniel P. Berrange ，[libvirt] [PATCH 0/8] Initial integration of lock managers，2010 年 11 月，https://www.redhat.com/archives/libvir-list/2010-November/msg00975.html

[8] Daniel P. Berrange，[libvirt] [PATCH 07/11] Add a 'nop' lock driver implementation，2011 年 01 月，https://www.redhat.com/archives/libvir-list/2011-January/msg00952.html

[9] Daniel P. Berrange，[libvirt] [PATCH 8/8] Add a plugin for the 'sanlock' project，2011 年 05 月，https://www.redhat.com/archives/libvir-list/2011-May/msg00630.html

[10] Daniel P. Berrange，[libvirt] [PATCH 00/14] Add a virtlockd lock manager daemon，2011 年 07 月，https://www.redhat.com/archives/libvir-list/2011-July/msg00337.html

[11] 周征晟，sanlock 原理介绍及应用——一种灵活轻量的分布式锁管理器，2014 年 04 月，https://www.ibm.com/developerworks/cn/linux/1404_zhouzs_sanlock/index.html

[12] Victor Gaydov，File locking in Linux，2016 年 07 月，https://gavv.github.io/blog/file-locks/

[13] Jeffrey T. Layton，File-private POSIX locks，LWN.net，2014 年 02 月

[14] Red Hat Bugzilla，Bug 1378241 - QEMU image file locking，2016 年 09 月，https://bu

gzilla.redhat.com/show_bug.cgi?id=1378241

[15] Tanja Roth，Thomas Schraitle，SUSE Linux Enterprise High Availability Extension 11 SP4 高可用性指南，2015 年 07 月，https://www.suse.com/zh-cn/documentation/sle_ha/singlehtml/book_sleha/book_sleha.html

[16] Corosync Project，Corosync Wiki FAQ，2012 年 09 月，https://github.com/corosync/corosync/wiki/FAQ

[17] Christine Caulfield，Programming Locking Applications，Red Hat Inc，2009 年 04 月

[18] Wikipedia，Kernel-based Virtual Machine，2018 年 03 月，https://en.wikipedia.org/wiki/Kernel-based_Virtual_Machine

[19] Wikipedia，libvirt，2018 年 02 月，https://en.wikipedia.org/wiki/Libvirt

[20] Libvirt Project，VM lifecycle，2013 年 07 月，https://wiki.libvirt.org/page/VM_lifecycle

[21] 刘世民，[译] libvirt 虚机的生命周期 （Libvirt Virtual Machine Lifecycle），2015 年 05 月，http://www.cnblogs.com/sammyliu/p/4486712.html

[22] Libvirt Project，Resource Lock Manager，2018 年 07 月，https://libvirt.org/internals/locking.html

[23] Red Hat Bugzilla，Bug 1337005 - Log event when a block device in use by a guest is open read-write by external applications ，2016 年 05 月，https://bugzilla.redhat.com/show_bug.cgi?id=1337005

[24] Fam Zheng，[Qemu-devel] [PATCH for-2.7 00/15] block: Lock images when opening，2016 年 04 月，https://lists.gnu.org/archive/html/qemu-devel/2016-04/msg02100.html