

**企业实习总结报告**

学号： 2014220301019

姓名： 卢杰

专业方向： 嵌入式系统

企业名称： 英特尔亚太研发有限公司

实习岗位名称： 虚拟化开发工程师

企业指导教师： 王震宇

院内代管教师： 廖勇

**信息与软件工程学院**

摘要

本文通过个人实习经历，阐述了实习所参与的项目中个人的贡献及学习到的知识经验。总结了测试、前端开发及页面设计的过程中遇到的复杂工程问题及相应解决方案。主要包括API测试、功能性能测试、页面UI设计及前端知识学习与应用。总结了我在实习中的收获，如知识技能的学习掌握情况、对工程管理、职业素养方面的认识等。

关键词：功能性能测试，API测试，UI设计

**ABSTRACT**

This report through personal practice experience, practice to participate in the project were introduced in detail in the individual contribution and learn knowledge. Through detailed project requirement analysis, carefully summarize the test, the front-end encountered in the process of development and design of complex engineering problems and corresponding solutions. Mainly includes the UI API testing, functional performance test, page design and front-end knowledge learning and application. On this basis, the introduction to the practice of the harvest, such as knowledge and skills of learning situation, the understanding of project management, professional quality, etc.

Keywords: GPU Container, Cgroup, Priority

目 录

[1. 企业实习的进展情况 2](#_Toc485223356)

[1.1. 实习工作完成情况 2](#_Toc485223357)

[1.1.1推理分析 2](#_Toc485223358)

[1.1.2总体设计 10](#_Toc485223359)

[1.1.3详细设计 11](#_Toc485223360)

[1.1.4 已完成工作 11](#_Toc485223361)

[1.1.5 分析与解释 12](#_Toc485223362)

[1.2. 知识技能学习情况 12](#_Toc485223363)

[1.2.1 开发环境说明 12](#_Toc485223364)

[1.2.2 工具学习使用 12](#_Toc485223365)

[1.2.3 其他学习内容 14](#_Toc485223366)

[1.3. 职业素养学习培养 14](#_Toc485223367)

[1.3.1 软件工程系统 14](#_Toc485223368)

[1.3.2 职业道德规范 15](#_Toc485223369)

[1.4. 工程协作与交流情况 16](#_Toc485223370)

[2. 存在问题与解决方案 17](#_Toc485223371)

[2.1. 存在的主要问题 17](#_Toc485223372)

[2.1.1 问题总结 17](#_Toc485223373)

[2.1.2 解决方式 17](#_Toc485223374)

[2.1.3 收获 18](#_Toc485223375)

[2.2. 解决方案与可行性研究 18](#_Toc485223376)

[2.2.1 尚未解决的问题 18](#_Toc485223377)

[2.2.2 解决方案 18](#_Toc485223378)

[2.2.3 可行性分析 18](#_Toc485223379)

[2.2.4 结论 21](#_Toc485223380)

[3. 前期任务完成度与后续实施计划 22](#_Toc485223381)

[3.1 前期任务完成度 22](#_Toc485223382)

[3.2 后续实施计划 22](#_Toc485223383)

[3.2.1 实施方案计划 22](#_Toc485223384)

[3.2.2 知识技能学习计划 22](#_Toc485223385)

[参考文献 23](#_Toc485223386)

# 企业实习概况

* 1. 实习单位情况和岗位责任
     1. 实习单位概况

英特尔是计算创新领域的全球领先厂商，设计和构建关键技术，为全球的计算设备奠定基础。1968年，罗伯特•诺伊斯(Robert•Noyce)、戈登•摩尔(Gordon Moore)和安迪•格鲁夫(Andy•Grove)在硅谷共同创立了英特尔公司。超过40年的发展，英特尔公司在芯片创新、技术开发、产品与平台等领域奠定了全球领先的地位，并始终引领着相关行业的技术产品创新及产业与市场的发展。英特尔的愿景是：未来十年创新和扩展计算技术，连接世界上每一个人，让生活变得丰富多彩。英特尔为计算机工业提供关键元件，包括性能卓越的微处理器、芯片组、系统及软件等，这些产品是标准计算机架构的重要组成部分。

英特尔1985年在北京设立代表处。今天英特尔公司已经成为在华最大的外国投资企业之一，员工人数超过7,500人。中国也是英特尔在美国本土以外部署最全面的国家，包括了芯片制造、封装测试，新技术研究、产品开发及营销——英特尔大连芯片厂是英特尔亚洲首家晶圆制造厂，英特尔成都工厂是世界一流的芯片封装测试厂，位于北京的英特尔中国研究院专注于前瞻性嵌入式系统研究，位于上海的英特尔亚太研发中心则是英特尔在中国的研发基地和创新中心。

其中英特尔亚太区研发有限公司于 2005 年 9 月在上海市紫竹科学园区正式成立——作为一个职能完备的研发机构，它兼具先进产品的开发能力和市场推广能力，将为中国及全球提供创新产品，为客户提供全面支持。英特尔亚太区研发有限公司主要从事高科技信息和通讯领域内的研究开发，同时参与中试和研发成果的转让及为英特尔产品提供相应技术服务。致力于推进围绕所有英特尔平台事业部的主要产品与技术的开发与创新。这些部门包括软件服务部、新设备部。

* + 1. 实习岗位职责

利用英特尔硬件平台，搭建虚拟化软件环境，与英特尔软件工程师一起从事Linux内核，设备驱动开发以及调试工作。

* + 1. 所属行业认识

众所周知，虚拟化技术是云计算的基础，对于这项工作所要求的个人能力和经验还是比较多的，所以在整个行业来看各家对这部分的投入和开发还处于一个饱满阶段。

* 1. 实习课题背景、价值、意义与国内外研究现状
     1. 课题背景、价值、意义

本课题来源于目前的很火的容器技术，出发点在于基于容器技术的大量，轻便，开销少的特点，完成其对GPU资源的支持，以便使用者结合容器技术进行Deep Learning，Machine Learning等方面的开发。由于目前公司力推Deep Learning，Machine Learning等人工智能前沿技术，所以为其提供基本的技术支持就成就了本项目的潜在的应用价值。

* + 1. 国内外研究现状

Nvidia 研究比较早，但是还是存在不足，其主要不足在于每个Nvidia container是独占整个GPU资源用于支持其cuda等的机器学习框架，缺乏更细粒度的GPU资源控制，而我们所做的是要基于cgroup机制，提供一个较为完善的GPU cgroup控制方案，为intel机器学习方案提供相关资源支持。

1.3 实习项目整体执行位完成情况概述

# 复杂工程问题归纳与实施方案可行性研究

* 1. 需求分析与实习目标任务
     1. 需求分析

当前容器技术的火热，其\*\*\*\*\*\*等特点为现实应用提供了更加方便和更低开销的选择，而GPU资源作为一种重要的硬件资源，既可以用于图像视频的支持，也广泛用于人工智能等领域的计算。经过我们的分析，目前市面上缺乏对容器技术与GPU资源结合的产品，作为GPU显示方面的巨头Nvidia也仅仅支持在Docker Container里提供GPU资源的基本使用以及Driver 版本的检测，据调查分析，市面上目前还没有能够对GPU资源容器化进行更细粒度控制与分配的方案。鉴于突出的市场需求，基于Docker Container的GPU Container也就应运而生了。

* + 1. 课题目标与任务

整个实习阶段最终目标为GPU Container POC，完成Plugin，能够基于Cgroup将Host的GPU resource 根据一定策略分配给多个Container使用。

* + 1. 实习任务

1. 了解XenGT/KVMGT project technology
2. GVT-g Upstream的编译安装测试
3. MediaSDK performance in Docker Container (Docker)
4. MediaSDK performance in Clear Container (intel)
5. Docker Plugin to assign GPU resource to Container
   * 1. 复杂性问题分析

操作系统领域一直以来面临的一个主要挑战来自于应用程序间存在的相互独立性和资源互操作性之间的矛盾，即每一个应用程序都希望运行在一个相对独立的系统环境下，不受到其他程序的干扰，同时又能以方便快捷的方式与其他程序交换和共享系统资源。当前面向个人计算机的通用操作系统更强调程序间的互操作性，而缺乏对程序间相互独立性的有效支持。

Vmware和Xen等虚拟化主流产品均采用Hypervisor模型，该模型通过将应用程序运行在多个不同虚拟机内，实现对上层应用程序的隔离。但是由于Hypervisor模型更倾向于每个虚拟机都拥有一份相对独立的系统资源，提供相对完全的独立性支持，这种策略造成处于不同虚拟机内的应用程序间实现互操作非常困难。

Hypervisor模型这种强独立性保障策略在一定程度上牺牲系统的执行效率。对于高性能计算，Web服务，数据库，游戏平台和分布式系统等许多应用领域，提供高效的资源互操作性同保持程序间的独立性具有同等重要的意义。因此，针对这样的需求，提出了一种基于资源和安全容器概念的虚拟化技术。

对于GPU这样一类特定资源的分配，就需要加入对容器虚拟化的支持。在intel公司有对应的Clear Container技术，外部有当前流行的Docker Container技术，结合Docker的Plugin技术来思考如何添加对于GPU resource的管理。在实现之前需要分析多个Container对于GPU resource的争夺对于各自的性能的影响。在分析基础上可以先从实现最基本assign一些资源给Container，再考虑添加一些调度策略进行GPU resource调度。此外，未来还需要添加一些接口方便供使用者调用。

* 1. 解决方案及可行性研究

**（1）Container与VM比较**

VM的理念是在宿主的系统上，自己虚拟了一个硬件平台，然后运行一个不同的OS，每个vm需要很多资源，所以一台机器上跑不了几个vm。

Container的理念是依托于宿主机提供的内核，仅仅把一个不同的Linux发行版所需的特性打包成一个image，当运行一个ubuntu镜像，感觉就是在一个ubuntu操作环境里，但实际调用的系统接口都来自宿主机。因为共用宿主机内核，所以其所需资源很少，性能开销很小，一台机器可以起上千个Container。其比较如图1-1所示。

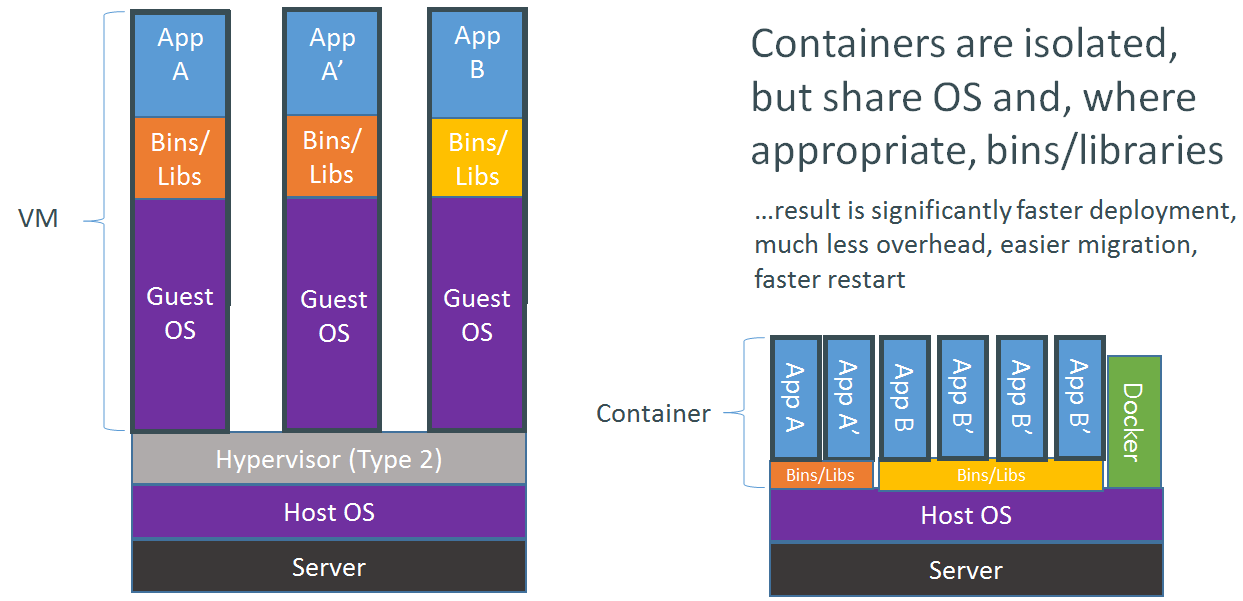


图1-1 Container与VM比较

表1-1 VM与Container关注点

|  |  |  |  |
| --- | --- | --- | --- |
|  | 运行在 | 关注 | 涉及的产品 |
| VM | Root Image运行于设备存储上；  Image存储在对象存储上；  文件存储不在VM关注层次上； | 性能  容量  可拓展 | VSAN，Ceph  存储接口RBD,  Cinder等 |
| Docker Container | Root Image运行于块设备存储  Image存储在对象存储上  Volume是数据存储的接口 | 文件分层  迁移  应用感知  细粒度控制 | Union FS  DeviceMapper  ZFS  存储Flocker |

**（2）Container组成**

一个Container大体上由namespaces，resource control，filesystem组成，其中namespaces用于限制使用对象所能看见的空间，resource control用于限制使用对象能够使用的资源多少；filesystem的上层用于保存当前的修改，底层用于读取image。一个Container的组成如图1-2所示。

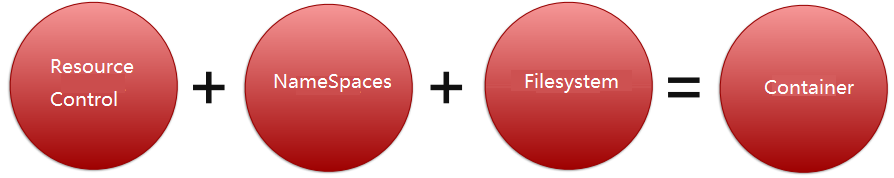


图1-2 Container基本组成要素

**（3）Container cgroup**

cgroup 是 Linux 内核的一个特性，此特性可以帮助用户对一组进程进行资源使用的限制、统计以及隔离。 Docker 领域也不例外， Docker 利用对 cgroup 的支持，完成对 Docker 容器进程组的资源限制、统计以及隔离。与 cgroup 内容相关的 Cgroup 定义，位于 ./docker/libcontainer/cgroups/cgroups.go，如下:

代码1-1 docker 的cgroup定义

|  |
| --- |
| type Cgroup struct {  Name string  Parent string  AllowAllDevices bool  // name of parent cgroup or slice  // If this is true allow access to any  kind of device within the container. If false,  allow access only to devices explicitly listed  in the allowed devices list.  AllowedDevices []\*devices.DeviceMemory int64  MemoryReservat int64  MemorySwap int64  CpuShares int64  CpuQuota int64  CpuPeriod int64  CpusetCpus string  //Memory limit (in bytes)  //Memory reservation or soft\_limit (in bytes)  //Total memory usage (memory + swap);  //CPU shares (re1ative weight vs. other containers)  //CPU hardcap limit (in usecs). Allowed cpu time in a given period.  //CPU period to be used for hardcapping (in usecs). 0 to use system default.  // CPU to use Freezer FreezerState  // set the freeze value for the process  // Parent slice to use for systemd |

从结构体 Cgroup 的定义可见， Docker 对于 cgroup 的支持，主要有以下 5 方面:设备( device)、内存 (Memo可)、 CPU 、 Freezer 以及 systemd。在容器设备方面， Docker 支持让用户选择容器可以使用的设备。在内存方面，支持为容器的运行设定用量限额。在 CPU 方面，支持容器进程之间拥有相对的运行时间片。 Freezer 则可以使容器挂起，节省 CPU 资源，Docker 命令中 pause 与 unpause 命令即采用了 cgroup 的 Freezer 子系统。需要注意的是，容器进程组挂起，并不意味着进程已经终止，从 Linux 内核的角度来看，被挂起的进程拥有完整的 task struct ，占用相应的内存，但是 Freezer 子系统保证容器中的进程不会被 CPU 调度到。 Slice 属性则属于 systemd 方面的配置。而对于GPU的支持则还没有实现，这也是我们工作的重点之一。

**（4）volume plugin**

Volume是个interface，Docker实现两种Volume：

1. 基于主机文件系统;

②基于Volume Plugin。

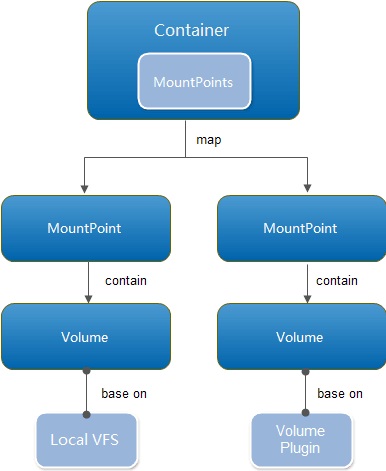


图1-3 Container中的Volume

Docker Volume Plugin的总体框架主要包括Docker Daemon对Volume的管理；docker volume的管理；docker plugin实现原理三个主要部分，通过分析其总体框架，得以分析plugin机制是否能够满足我们对于driver版本检查的目的。

1. **Docker Daemon对Volume的管理**

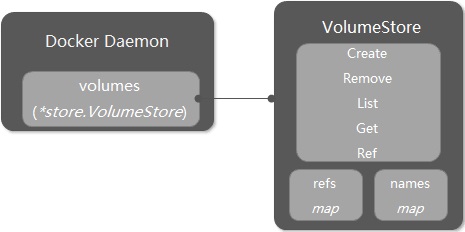


图1-4 Daemon 管理 Volume

如上图1-4所示，Docker Daemon结构中有个成员Volumes，类型是VolumeStore类型，包含一组管理Volume的函数：Create、Remove、List、Get、Refs ...。通过两个变量，管理Container和Volume的关系。 names : map结构，Key是Volume的name，value是实现Volume接口的结构对象。存储该Daemon内所有的Volume。 ReFS: map结构，key是volume的name，value是string数组保存引用该Volume的Container ID。 Docker Daemon通过Volumes变量，就可以管理所有的Volume，提供如下命令：

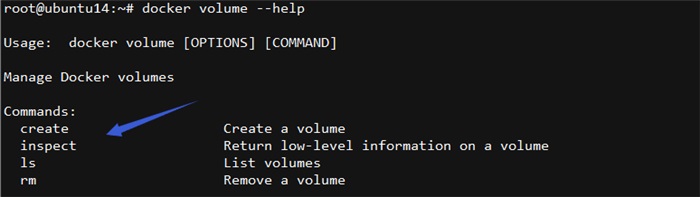


图1-5 Volume 命令

1. **docker volume管理**

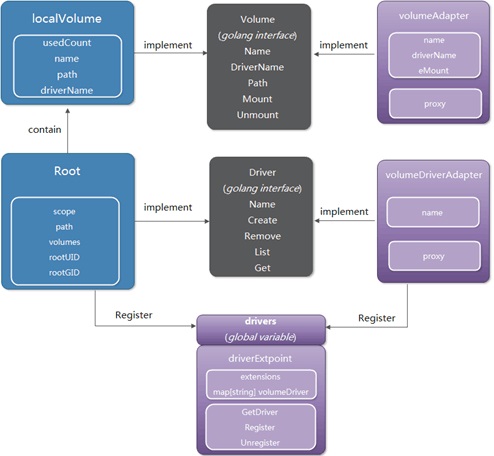


图1-6 基于本地文件系统的volume框架

Docker提供两个接口Volume和Driver，所有提供给Docker使用的Volume必须实现Volume接口。后端驱动需要实现Driver接口。Driver是对提供出去的Volume进行管理。目前Docker实现了两种Volume &Driver。

1. **基于本地文件系统的Volume**

可以在执行Docker create或Docker run时，通过-v参数将主机的目录作为容器的数据卷。这部分功能便是基于本地文件系统的volume管理。上图中蓝色部分LocalVolume和Root。这两个结构就是对主机目录和文件进行管理，具体的对应关系如下图：

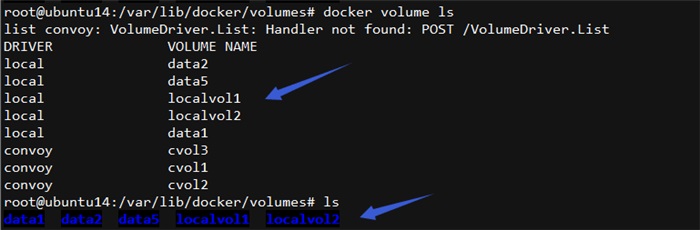


图1-7 localvolume 管理

从上图1-7可以看出，基于本地文件系统的卷管理，Driver便是卷的根目录/var/lib/docker/volumes。一个卷就是根目录下的一个子目录。

1. **适配Plugin的Volume Docker**

为了支持第三方存储方案，在1.8版本引入Volume Plugin机制，Volume Adapter和Volume Driver Adapter分别实现了接口Volume和Driver接口，用于表示由Plugin提供的Volume和Plugin driver。在下一小节详细描述。

基于本地文件系统的Volume框架中，全局变量drivers保存了所有注册到Docker Daemon的Driver。Docker Daemon需要对Volume进行管理操作时，通过GetDriver函数从drivers变量中获取指定名称的Driver，通过Driver可以通过Create，Remove，List，Get对Driver中Volume进行管理。通过Get函数获取Volume结构，可以对卷进行管理操作：Name，DriverName，Path，Mount，Unmount。

1. **docker plugin实现原理**

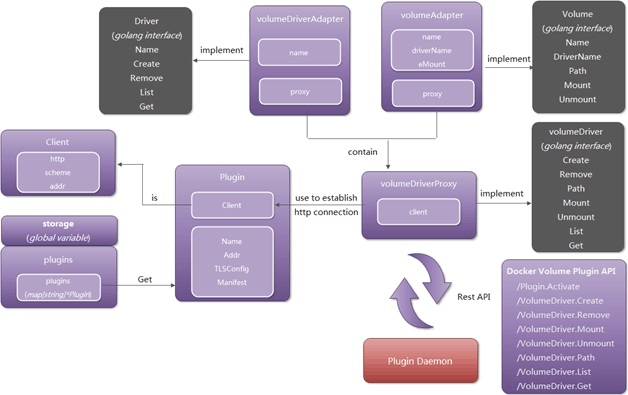


图1-8 Volume Plugin实现原理

1. **Docker Plugin机制**

Docker针对Volume Plugin实现了两个适配类型Volume Driver Adapter和Volume Adapter。

Volume Driver Adapter : 实现Driver接口，用于抽象各种Plugin的驱动，该类型可以适配所有符合规范的Volume Plugin，对Plugin进行管理。

Volume Adapter : 实现Volume接口，用于抽象所有Plugin提供的Volume，该类型可以适配所有符合规范的Volume Plugin提供的类型，对Volume进行管理。 通过抽象，对于Plugin和其提供的Volume，Docker Daemon结构和Container结构使用上述两个适配类型，对Plugin和Volume进行管理和使用。

上述两个类型都有一个Volume Driver Proxy结构变量proxy，用于与Plugin进行通信。Volume Driver Proxy结构实现了与plugin通信的接口volume Driver，提供Create、Remove、Path、Mount、Unmount、List、Get接口与通信。Volume Driver Proxy结构的client变量，使用这个变量与Plugin Daemon进行通信。client变量是Plugin结构中的Client变量是Client结构类型，包含了http.Client类型对象。

图1-8 中左下方，Plugins结构类型的全局变量Storage保存所有被Docker Daemon发现的Plugin结构。

1. **Docker Plugin的发现过程**

Docker Daemon通过Unix域套接字与Plugin Daemon进行通信。所以Plugin需要让Docker Daemon知道Plugin的Unix域套接字文件的路径。再执行命令：docker run .-v volumename:/data --volume-driver=convoy

发现步骤： Docker Daemon首先会在/run/docker/plugins搜索对应的套接字文件，套接字文件名必须和Volume Driver名一致，如上述命令，便是搜索convoy.sock。 如果上一步搜索不到，则到/etc/docker/plugins和/usr/lib/docker/plugins两个目录搜索spec或json文件。文件中指定套接字文件的URL。如：unix:///var/run/convoy/convoy.sock 。 根据前面两步发现的Unix域套接字URL，构建Plugin对象，并将新建对象加入到全局变量storage的Plugins字段中。

**（5）docker 性能测试**

下面是我用mediaSDK软件测试的关于docker container性能的数据柱状图，有助于判断采用docker container方案的一个性能的变化趋势。

图1-9 docker container性能分析

如上图1-9所示，在创建1个容器时，container的性能接近宿主机转解码性能；随着创建的container数量的增多，其性能与宿主机相比会有一个下降，但是相对还是一个比较合理的性能区间，这一点可以从图中两个柱状图的高度差的扩大可以发现。通过这些数据可以发现采用docker的方案是可行的。

* 1. 课题任务详细需求分析

# 3. 针对复杂工程问题的设计与实现

## 3.1. 针对复杂工程问题展开设计

### 3.1.2总体设计

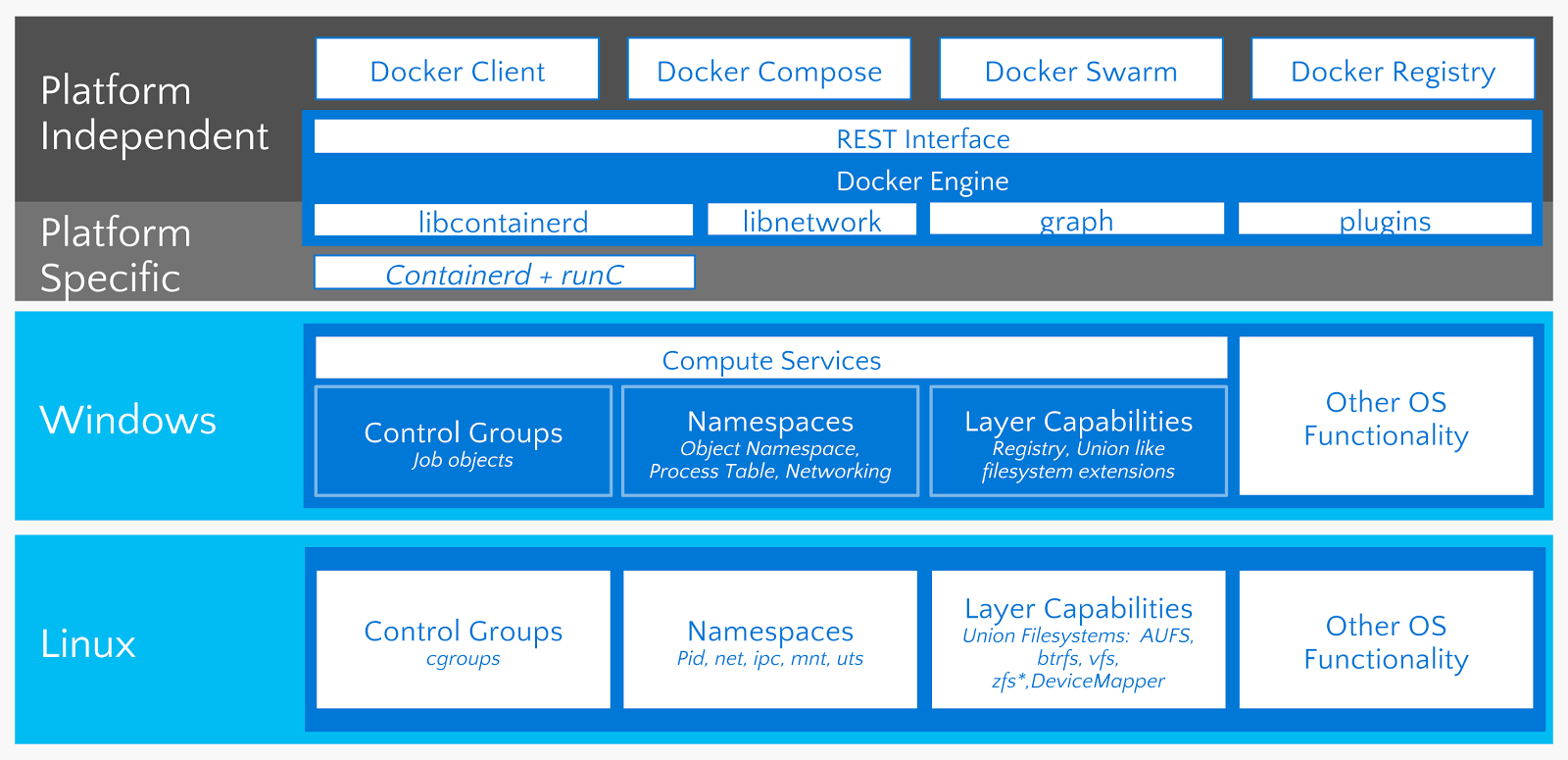


图1-10 总体架构图

Libcontainer提供原生go去用namespaces，cgroup，capabilities和filesystem access control去创建一个container。RunC在运行时是一个轻量级通用容器，提供一个命令行界面，根据OCI规范运行容器。Containerd对于容器来说是作为一个daemon，管理容器在宿主机系统下的整个生命周期的活动，比如image 传输和存储，容器创建和挂起等。Docker是一个用于build，ship和run container的一个开源产品。其流程如下图1-11所示。

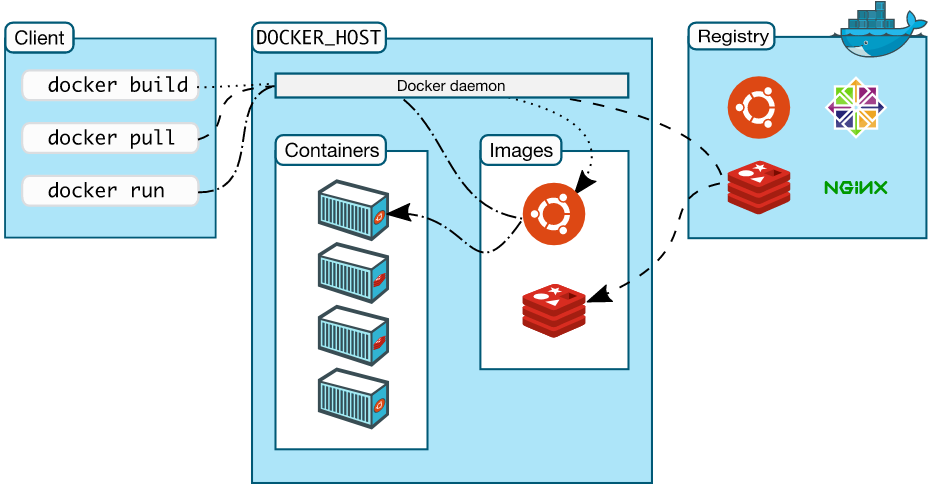


图1-11 client/server架构

添加kernel层的GPU memory和priority cgroup的控制，这是从系统层角度需要的设计；同时上层的Docker Client，libcontainer和RunC也需要添加。

### 3.1.3详细设计

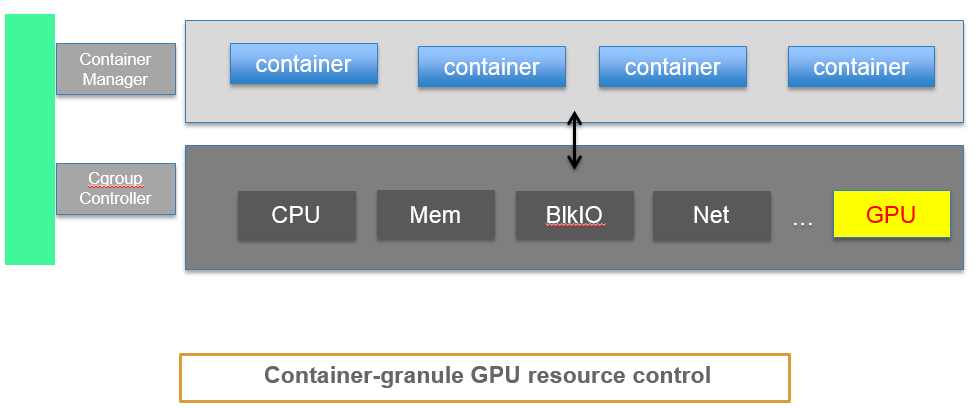


图1-12 详细设计图

在Docker所支持的从group类别基础上再添加针对GPU资源的控制，包括对kernel的GPU cgroup的添加和Docker Daemon及CLI的cgroup和参数添加。具体设计如下：

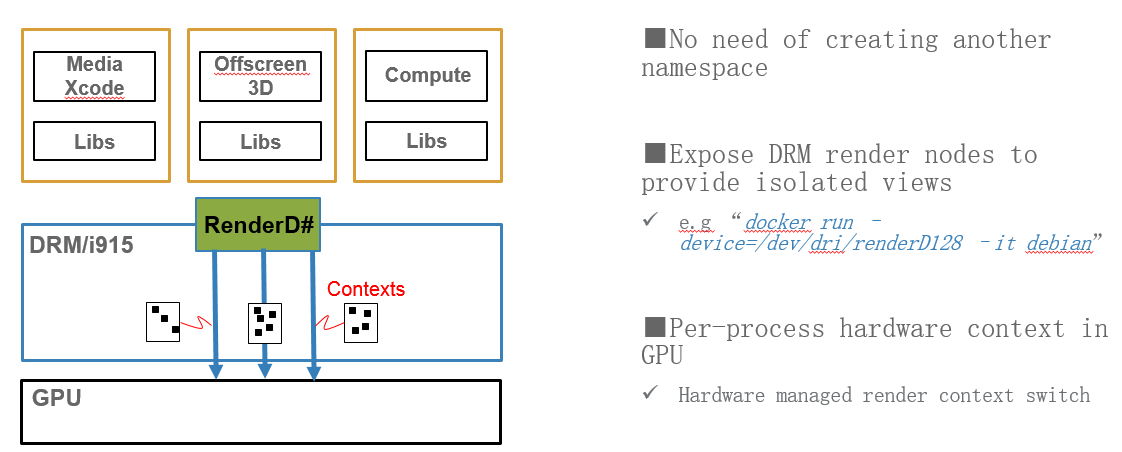


图1-13 资源控制层次图

需要一个新的子系统，gpu\_subsys，进行cgroup控制主要在三个方面需要，使用GPU百分比，工作任务的优先级以及最大分配的内存。在DRM/I915层需要支持任务调度过程中的GPU分配百分比和优先级，设定内存最大限制等。上层RunC里需要新的config.json文件，在该文件里面添加上关于GPU的参数定义。

### 3.1.4 已完成工作

1. 在kernel cgroup里添加GPU资源控制，包括GPU priority和最大memory控制；

|  |
| --- |
| diff --git a/drivers/gpu/drm/drm\_gem.c b/drivers/gpu/drm/drm\_gem.c  index b1e28c9..11a207e 100644  --- a/drivers/gpu/drm/drm\_gem.c  +++ b/drivers/gpu/drm/drm\_gem.c  @@ -775,8 +775,12 @@ drm\_gem\_object\_release(struct drm\_gem\_object \*obj)  {  WARN\_ON(obj->dma\_buf);  - if (obj->filp)  + if (obj->filp) {  + struct drm\_file \*file\_priv = obj->filp->private\_data;  + if (file\_priv)  + drm\_gem\_obj\_del\_mem(file\_priv, obj->size);  fput(obj->filp);  + }  drm\_gem\_free\_mmap\_offset(obj);  }  @@ -1004,3 +1008,40 @@ int drm\_gem\_mmap(struct file \*filp, struct vm\_area\_struct \*vma)  return ret;  }  EXPORT\_SYMBOL(drm\_gem\_mmap);  +  +int drm\_gem\_obj\_check\_max\_mem(struct drm\_file \*file\_priv, u64 size)  +{  + int ret = 0;  + struct task\_struct \*task;  +  + spin\_lock(&file\_priv->obj\_stat\_lock);  +  + task = get\_pid\_task(file\_priv->pid, PIDTYPE\_PID);  + if (task) {  + if (file\_priv->obj\_mem + size > gpucg\_get\_max\_mem(task)) {  + ret = 1;  + DRM\_DEBUG\_DRIVER("ZHEN: hit max mem %d\n", task->pid);  + }  + put\_task\_struct(task);  + }  + spin\_unlock(&file\_priv->obj\_stat\_lock);  +  + return ret;  +}  +EXPORT\_SYMBOL(drm\_gem\_obj\_check\_max\_mem);  +  +void drm\_gem\_obj\_add\_mem(struct drm\_file \*file\_priv, u64 size)  +{  + spin\_lock(&file\_priv->obj\_stat\_lock);  + file\_priv->obj\_mem += size;  + spin\_unlock(&file\_priv->obj\_stat\_lock);  +}  +EXPORT\_SYMBOL(drm\_gem\_obj\_add\_mem);  +  +void drm\_gem\_obj\_del\_mem(struct drm\_file \*file\_priv, u64 size)  +{  + spin\_lock(&file\_priv->obj\_stat\_lock);  + file\_priv->obj\_mem -= size;  + spin\_unlock(&file\_priv->obj\_stat\_lock);  +}  +EXPORT\_SYMBOL(drm\_gem\_obj\_del\_mem); |

|  |
| --- |
| diff --git a/drivers/gpu/drm/i915/i915\_gem.c b/drivers/gpu/drm/i915/i915\_gem.c  index 33fb11c..6566ff5 100644  --- a/drivers/gpu/drm/i915/i915\_gem.c  +++ b/drivers/gpu/drm/i915/i915\_gem.c  @@ -657,6 +657,9 @@ i915\_gem\_create(struct drm\_file \*file,  if (size == 0)  return -EINVAL;  + if (drm\_gem\_obj\_check\_max\_mem(file, size))  + return -ENOMEM;  +  /\* Allocate the new object \*/  obj = i915\_gem\_object\_create(dev\_priv, size);  if (IS\_ERR(obj))  @@ -668,6 +671,7 @@ i915\_gem\_create(struct drm\_file \*file,  if (ret)  return ret;  + drm\_gem\_obj\_add\_mem(file, size);  \*handle\_p = handle;  return 0;  } |

|  |
| --- |
| diff --git a/drivers/gpu/drm/i915/i915\_gem\_context.c b/drivers/gpu/drm/i915/i915\_gem\_context.c  index 8bd0c49..7c7109b 100644  --- a/drivers/gpu/drm/i915/i915\_gem\_context.c  +++ b/drivers/gpu/drm/i915/i915\_gem\_context.c  @@ -377,6 +377,8 @@ i915\_gem\_create\_context(struct drm\_i915\_private \*dev\_priv,  ctx->desc\_template = default\_desc\_template(dev\_priv, ppgtt);  }  + ctx->priority = gpucg\_get\_priority(current);  +  trace\_i915\_context\_create(ctx);  return ctx; |

|  |
| --- |
| diff --git a/drivers/gpu/drm/i915/i915\_gem\_execbuffer.c b/drivers/gpu/drm/i915/i915\_gem\_execbuffer.c  index af19657..a389271 100644  --- a/drivers/gpu/drm/i915/i915\_gem\_execbuffer.c  +++ b/drivers/gpu/drm/i915/i915\_gem\_execbuffer.c  @@ -1648,6 +1648,9 @@ i915\_gem\_do\_execbuffer(struct drm\_device \*dev, void \*data,  i915\_gem\_context\_get(ctx);  + /\* a little hack for cg prio setting anyway \*/  + ctx->priority = (int)gpucg\_get\_priority(current);  +  if (ctx->ppgtt)  vm = &ctx->ppgtt->base;  else |

|  |
| --- |
| /\*  \* GPU cgroup  \*  \* GPU resource control based on cgroup hierarchy  \*  \*  \*/  #include <linux/slab.h>  #include <linux/cgroup.h>  #include <linux/cgroup\_gpu.h>  struct gpu\_cgroup {  struct cgroup\_subsys\_state css;  s64 prio;  u64 max\_mem\_in\_bytes;  };  static DEFINE\_MUTEX(gpucg\_mutex);  #define GPU\_PRIO\_MAX 1024 /\* align with i915 now \*/  static struct gpu\_cgroup \*css\_gpucg(struct cgroup\_subsys\_state \*css)  {  return container\_of(css, struct gpu\_cgroup, css);  }  static inline struct gpu\_cgroup \*get\_current\_gpucg(void)  {  return css\_gpucg(task\_get\_css(current, gpu\_cgrp\_id));  }  static inline struct gpu\_cgroup \*get\_task\_gpucg(struct task\_struct \*task)  {  return css\_gpucg(task\_get\_css(task, gpu\_cgrp\_id));  }  static s64 gpu\_css\_get\_prio(struct cgroup\_subsys\_state \*css, struct cftype \*cft)  {  struct gpu\_cgroup \*cg = css\_gpucg(css);  return cg->prio;  }  static int gpu\_css\_set\_prio(struct cgroup\_subsys\_state \*css, struct cftype \*cft,  s64 val)  {  struct gpu\_cgroup \*cg = css\_gpucg(css);  if (val > GPU\_PRIO\_MAX || val < -GPU\_PRIO\_MAX)  return -EINVAL;  mutex\_lock(&gpucg\_mutex);  cg->prio = val;  mutex\_unlock(&gpucg\_mutex);  return 0;  }  static u64 gpu\_css\_get\_max\_mem(struct cgroup\_subsys\_state \*css, struct cftype \*cft)  {  struct gpu\_cgroup \*cg = css\_gpucg(css);  return cg->max\_mem\_in\_bytes;  }  static int gpu\_css\_set\_max\_mem(struct cgroup\_subsys\_state \*css, struct cftype \*cft,  u64 val)  {  struct gpu\_cgroup \*cg = css\_gpucg(css);  mutex\_lock(&gpucg\_mutex);  cg->max\_mem\_in\_bytes = val;  mutex\_unlock(&gpucg\_mutex);  return 0;  }  static struct cftype gpu\_css\_files[] = {  {  .name = "priority",  .write\_s64 = gpu\_css\_set\_prio,  .read\_s64 = gpu\_css\_get\_prio,  .flags = CFTYPE\_NOT\_ON\_ROOT,  },  {  .name = "max\_mem\_in\_bytes",  .write\_u64 = gpu\_css\_set\_max\_mem,  .read\_u64 = gpu\_css\_get\_max\_mem,  .flags = CFTYPE\_NOT\_ON\_ROOT,  },  };  s64 gpucg\_get\_priority(struct task\_struct \*task)  {  struct gpu\_cgroup \*cg = get\_task\_gpucg(task);  BUG\_ON(!cg);  return cg->prio;  }  EXPORT\_SYMBOL(gpucg\_get\_priority);  u64 gpucg\_get\_max\_mem(struct task\_struct \*task)  {  struct gpu\_cgroup \*cg = get\_task\_gpucg(task);  BUG\_ON(!cg);  return cg->max\_mem\_in\_bytes;  }  EXPORT\_SYMBOL(gpucg\_get\_max\_mem);  static struct cgroup\_subsys\_state \*  gpu\_css\_alloc(struct cgroup\_subsys\_state \*parent\_css)  {  struct gpu\_cgroup \*cg;  cg = kzalloc(sizeof(\*cg), GFP\_KERNEL);  if (!cg)  return ERR\_PTR(-ENOMEM);  cg->prio = 0;  cg->max\_mem\_in\_bytes = ULONG\_MAX;  return &cg->css;  }  static void gpu\_css\_free(struct cgroup\_subsys\_state \*css)  {  struct gpu\_cgroup \*cg = css\_gpucg(css);  kfree(cg);  }  struct cgroup\_subsys gpu\_cgrp\_subsys = {  .css\_alloc = gpu\_css\_alloc,  .css\_free = gpu\_css\_free,  .legacy\_cftypes = gpu\_css\_files,  .dfl\_cftypes = gpu\_css\_files,  }; |

1. 在docker server端添加GPU资源控制；

需要在host\_config.go里添加关于GPU的变量

|  |
| --- |
| type Resources struct {  OomKillDisable \*bool // Whether to disable OOM Killer or not  PidsLimit int64 // Setting pids limit for a container  Ulimits []\*units.Ulimit // List of ulimits to be set in the container  + GPUMemory uint64 // Maximum GPU memory used by the container  + GPUPriority int64 // Priority set for GPU in container |

在Docker Daemon端daemon\_unix.go添加函数用于获取GPU资源：

|  |
| --- |
| func getGPUResources(config containertypes.Resources) \*specs.GPU {  gpu := specs.GPU{}  if config.GPUMemory != 0 {  period := uint64(config.GPUMemory)  gpu.Memory = &period  }  if config.GPUPriority != 0 {  runtime := int64(config.GPUPriority)  gpu.Priority = &runtime  }  return &gpu  } |

在config.go中添加相关配置：

|  |
| --- |
| +// GPU for Linux cgroup 'gpu' resource management  +type GPU struct {  + // GPU Mem Limit  + Memory \*uint64 `json:"memory,omitempty"`  + // GPU Priority  + Priority \*int64 `json:"prio,omitempty"`  }  // Resources has container runtime resource constraints  type Resources struct {  HugepageLimits []HugepageLimit `json:"hugepageLimits,omitempty"`  // Network restriction configuration  Network \*Network `json:"network,omitempty"`  + // GPU  + GPU \*GPU `json:"gpu,omitempty"`  } |

此外，docker的编译有一套较为特别的编译方式，这是由其庞大的package依赖所导致的，我们需要采用Dockerfile来完成我们修改之后的docker编译过程：

|  |
| --- |
| # CLI  +DOCKERCLI\_REPO=https://github.com/intel/cli.git    install\_runc() {  echo "Install runc version $RUNC\_COMMIT"  + git clone https://github.com/intel/runc.git "$GOPATH/src/github.com/opencontainers/runc"  cd "$GOPATH/src/github.com/opencontainers/runc"  + git checkout gpu-cgroup  make BUILDTAGS="$RUNC\_BUILDTAGS" $1  cp runc /usr/local/bin/docker-runc  }  install\_dockercli() {  echo "Install docker/cli version $DOCKERCLI\_COMMIT"  git clone "$DOCKERCLI\_REPO" "$GOPATH/src/github.com/docker/cli"  cd "$GOPATH/src/github.com/docker/cli"  + git checkout gpu-cgroup  go build -o /usr/local/bin/docker github.com/docker/cli/cmd/docker  } |

1. 在RunC里添加GPU资源控制；

在opencontainer里面的config.go文件添加关于GPU配置支持：

|  |
| --- |
| // GPU for Linux cgroup 'gpu' resource management  type GPU struct {  // GPU Mem Limit  Memory \*uint64 `json:"memory,omitempty"`  // GPU Priority  Priority \*int64 `json:"prio,omitempty"`  }  // Resources has container runtime resource constraints  type Resources struct {  HugepageLimits []HugepageLimit `json:"hugepageLimits,omitempty"`  // Network restriction configuration  Network \*Network `json:"network,omitempty"`  + // GPU resource restriction configuration  + GPU \*GPU `json:"gpu,omitempty"`  }  var (  &PerfEventGroup{},  &FreezerGroup{},  &NameGroup{GroupName: "name=systemd", Join: true},  + &GPUGroup{},  }  HugePageSizes, \_ = cgroups.GetHugePageSize()  ) |

在libcontainer的cgroup里新建关于GPU的基本操作接口函数：

|  |
| --- |
| // +build linux  package fs  import (  "fmt"  "os"  "strconv"  "github.com/opencontainers/runc/libcontainer/cgroups"  "github.com/opencontainers/runc/libcontainer/configs"  )  const (  cgroupGPUMemoryLimit = "gpu.max\_mem\_in\_bytes"  cgroupGPUPriority = "gpu.priority"  )  type GPUGroup struct {  }  func (s \*GPUGroup) Name() string {  return "gpu"  }  func (s \*GPUGroup) Apply(d \*cgroupData) (err error) {  path, err := d.path("gpu")  if err != nil && !cgroups.IsNotFound(err) {  return err  } else if path == "" {  return nil  }  fmt.Printf("gpu cgroup %s %d\n", path, d.pid)  if err := os.MkdirAll(path, 0755); err != nil {  return err  if err := cgroups.WriteCgroupProc(path, d.pid); err != nil {  return err  }  return nil  }  func (s \*GPUGroup) Set(path string, cgroup \*configs.Cgroup) error {  if cgroup.Resources.GpuMemory != 0 {  if err := writeFile(path, cgroupGPUMemoryLimit, strconv.FormatUint(cgroup.Resources.GpuMemory, 10)); err != nil {  return err  }  }  if cgroup.Resources.GpuPriority != 0 {  if err := writeFile(path, cgroupGPUPriority, strconv.FormatInt(cgroup.Resources.GpuPriority, 10)); err != nil {  return err  }  }  return nil  }  func (s \*GPUGroup) Remove(d \*cgroupData) error {  return removePath(d.path("gpu"))  }  func (s \*GPUGroup) GetStats(path string, stats \*cgroups.Stats) error {  return nil  }  func gpuAssigned(cgroup \*configs.Cgroup) bool {  return cgroup.Resources.GpuMemory != 0 ||  cgroup.Resources.GpuPriority != 0  } |

|  |
| --- |
| type Resources struct {  // Set class identifier for container's network packets  NetClsClassid uint32 `json:"net\_cls\_classid\_u"`  // GPU Memory limit (in bytes)  GpuMemory uint64 `json:"gpu\_memory"`  // GPU priority  GpuPriority int64 `json:"gpu\_prio"`  } |

在libcontainer的spec\_linux.go中添加创建GPU配置文件的函数：

|  |
| --- |
| func createCgroupConfig(name string, useSystemdCgroup bool, spec \*specs.Spec) (\*  })  }  }  if r.GPU != nil {  if r.GPU.Memory != nil {  c.Resources.GpuMemory = \*r.GPU.Memory  }  if r.GPU.Priority != nil {  c.Resources.GpuPriority = \*r.GPU.Priority  }  }  return c, nil  } |

|  |
| --- |
| import (  "github.com/urfave/cli"  )  +func i64Ptr(i int64) \*int64 { return &i }  func u64Ptr(i uint64) \*uint64 { return &i }  func u16Ptr(i uint16) \*uint16 { return &i }  other options are ignored.  BlockIO: &specs.BlockIO{  Weight: u16Ptr(0),  },  + GPU: &specs.GPU{  + Memory: u64Ptr(0),  + Priority: i64Ptr(0),  + },  }  config := container.Config()  other options are ignored.  }  for \_, pair := range []struct {  opt string  + dest \*int64  + }{  + {"gpu\_prio", r.GPU.Priority},  + } {  + if val := context.String(pair.opt); val != "" {  + var err error  + \*pair.dest, err = strconv.ParseInt(val, 10, 64)  + if err != nil {  + return fmt.Errorf("invalid value for %s: %s", pair.opt, err)  + }  + }  + }  + for \_, pair := range []struct {  + opt string  dest \*uint64  }{  {"kernel-memory", r.Memory.Kernel},  other options are ignored.  {"memory", r.Memory.Limit},  {"memory-reservation", r.Memory.Reservation},  {"memory-swap", r.Memory.Swap},  + {"gpu\_memory", r.GPU.Memory},  } {  if val := context.String(pair.opt); val != "" {  v, err := units.RAMInBytes(val)  @@ -221,6 +241,9 @@ other options are ignored.  config.Cgroups.Resources.MemoryReservation = int64(\*r.Memory.Reservation)  config.Cgroups.Resources.MemorySwap = int64(\*r.Memory.Swap)  + config.Cgroups.Resources.GpuMemory = \*r.GPU.Memory  + config.Cgroups.Resources.GpuPriority = \*r.GPU.Priority  +  if err := container.Set(config); err != nil {  return err  } |

1. 在docker client端的CLI添加GPU资源控制；

修改在cli端的container中的opts.go文件，添加关于GPU的命令参数：--gpu-priority和—gpu-memory为新定义的用户参数名称，用于设置容器的gpu的优先级和内存大小。

|  |
| --- |
| type containerOptions struct {  runtime string  autoRemove bool  init bool  + gpuMemory uint64  + gpuPriority int64  Image string  Args []string  }  func addFlags(flags \*pflag.FlagSet) \*containerOptions {  flags.BoolVar(&copts.init, "init", false, "Run an init inside the container that forwards signals and reaps processes")  flags.SetAnnotation("init", "version", []string{"1.25"})  + flags.Uint64Var(&copts.gpuMemory, "gpu-memory", 0, "Limit GPU memory")  + flags.Int64Var(&copts.gpuPriority, "gpu-priority", 0, "Limit GPU priority")  return copts  }  func parse(flags \*pflag.FlagSet, copts \*containerOptions) (\*containerConfig, err  Ulimits: copts.ulimits.GetList(),  DeviceCgroupRules: copts.deviceCgroupRules.GetAll(),  Devices: deviceMappings,  + GPUMemory: copts.gpuMemory,  + GPUPriority: copts.gpuPriority,  } |

1. Intel-docker-plugin核心接口实现；

获取并解析用户client端参数设置：

|  |
| --- |
| func docker(stdout bool, command string, arg ...string) (b []byte, err error) {  var buf bytes.Buffer  args := append(append(dockerCmd[1:], command), arg...)  cmd := exec.Command(dockerCmd[0], args...)  cmd.Stderr = &buf  if stdout {  cmd.Stdout = os.Stderr  err = cmd.Run()  } else {  b, err = cmd.Output()  }  if err != nil {  b = bytes.TrimSpace(buf.Bytes())  b = bytes.TrimPrefix(b, []byte("Error: "))  if len(b) > 0 {  return nil, fmt.Errorf("%s", b)  } else {  return nil, fmt.Errorf("failed to run docker command")  }  }  return b, nil  } |

获取本地驱动版本：

|  |
| --- |
| func GenerateLocalArgs(image string, vols []string) ([]string, error) {  cv, err := nvidia.GetCUDAVersion()  if err != nil {  return nil, err  }  if err := cudaSupported(image, cv); err != nil {  return nil, err  }  d, err := devicesArgs()  if err != nil {  return nil, err  }  v, err := volumesArgs(vols)  if err != nil {  return nil, err  }  return append(d, v...), nil  } |

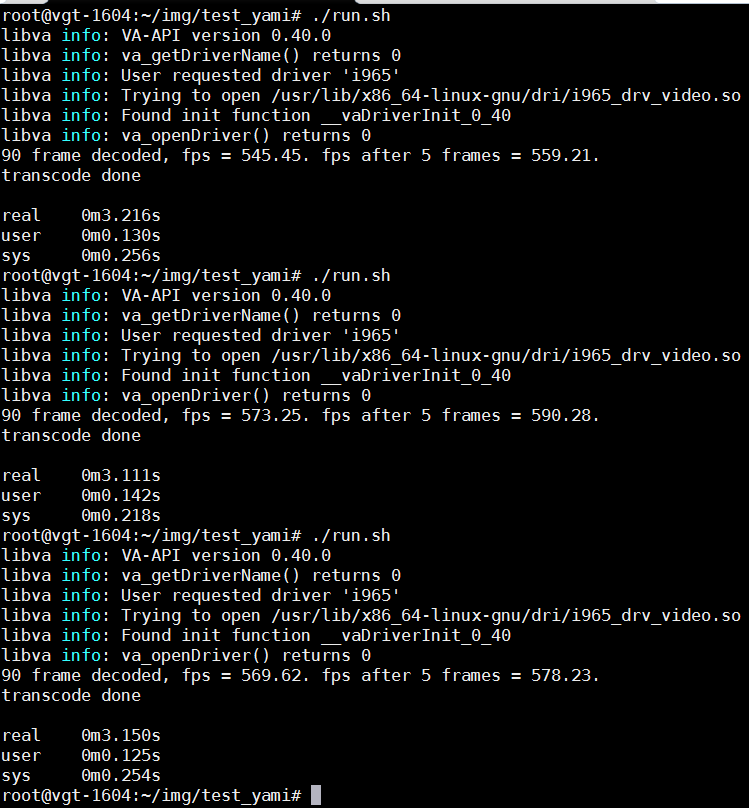
获取container image里面的版本标签：

|  |
| --- |
| func Label(image, label string) (string, error) {  format := fmt.Sprintf(`--format={{index .Config.Labels "%s"}}`, label)  b, err := docker(false, "inspect", format, image)  if err != nil {  return "", err  }  return string(bytes.Trim(b, " \n")), nil  } |

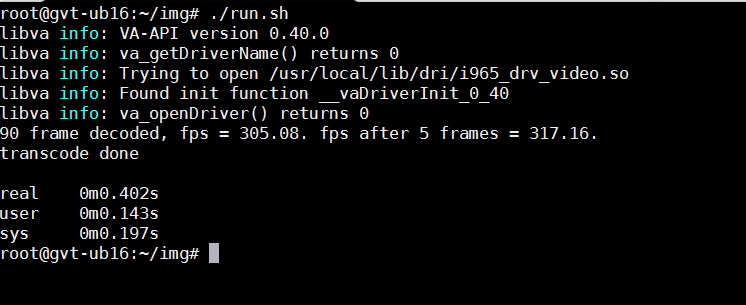
1. 编写关于用Docker在linux container里使能GPU的wiki；
2. 成功在container里面运行yami的测试样例。

I．如图1.1到1.3所示，分别为host跑yami的fps，在host上起的clear container的跑yami的fps以及在clear container中起一个docker container来跑yami的fps的比较。

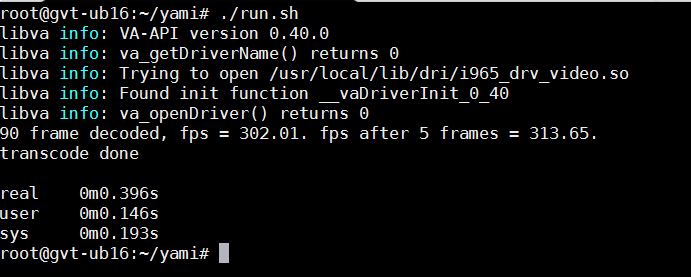
在宿主机上：



在宿主机上起的一个Clear Container中：



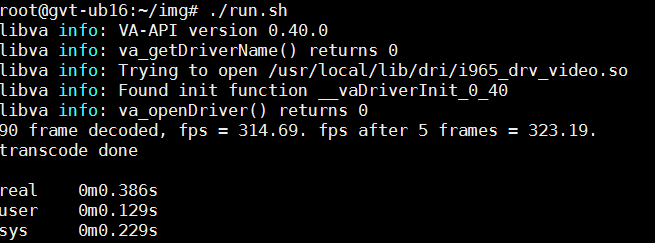
在Clear Container中再起一个Docker Container：



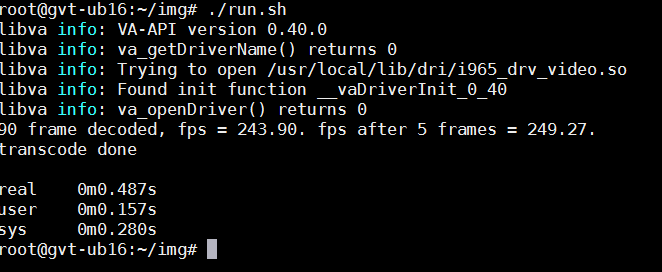
Host，Clear Container，Docker in container的性能对比如图1-1所示

II．分配不同vgpu的情况如图1.1-1.3所示：

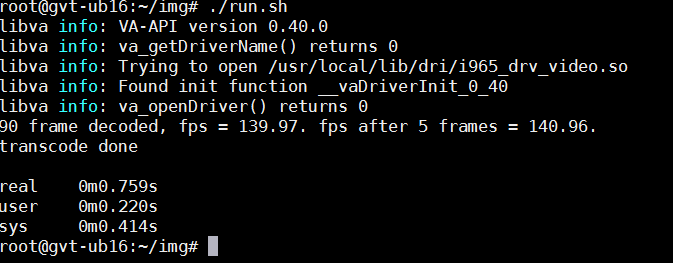
分成一个vgpu，全部分给这个container：



分成两个vgpu，分配二分之一给这个container：



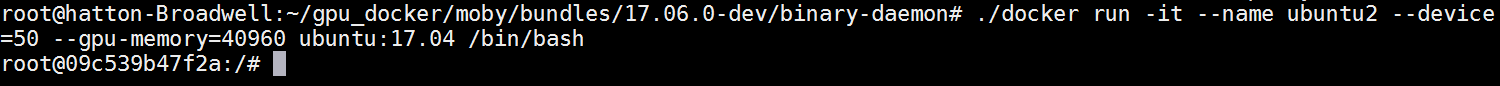
分成四个vgpu，分配四分之一给这个container：

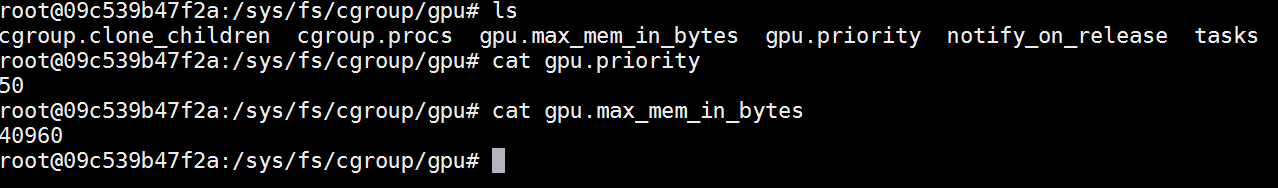
II．基于Cgroup控制的GPU Container的测试，可通过查看系统cgroup目录，发现已支持gpu，且对于不同的memory和priority的container，其cgroup中gpu目录里面所分配的资源值是不同的（--gpu-priority和—gpu-memory为我们新定义的使用参数，便于赋予container相应的值）。

分配memory和priority测试情况（分配priority为50，memory为10240）：

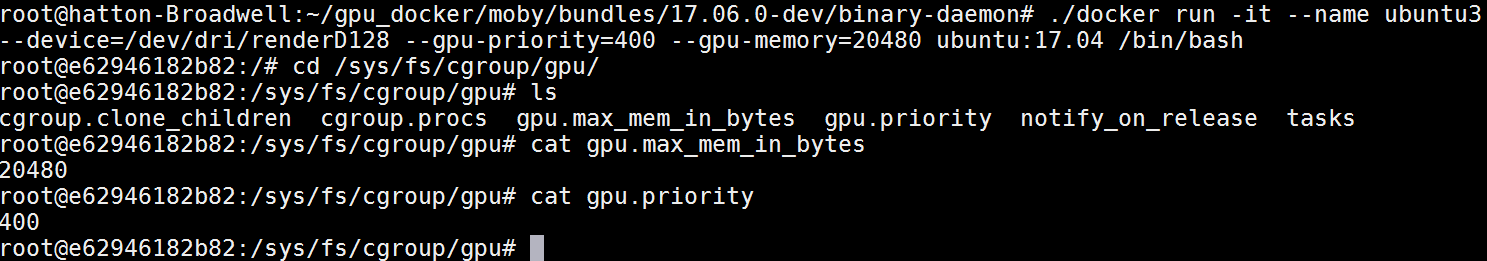


分配memory和priority测试情况（分配priority为50，memory为40960）：





分配memory和priority测试情况（分配priority为400，memory为20480）：



### 3.1.5 分析与解释

如果采用volume plugin来进行容器创建前的driver版本检查的话，有一个最直接的问题需要考虑的是volume主要的范围还是一个数据的持久化存储和数据迁移与共享，在运行时主要涉及的还是一个数据处理的过程，没有参与执行脚本的倾向。这对我们组采用volume plugin来进行拓展检查容器内driver版本的功能产生的困扰。

## 3.2. 知识技能学习情况

### 3.2.1 开发环境说明

本组项目的开发是Broadwell/Skylake server上进行的，CPU为i5-5300U @2.3GHz（Max Speed 2.9GHz），4Core，16G Mem，GPU 900MHz，Turbo Ratio 57；软件环境为自己编译的ubuntu 4.12.0-rc4+版本；Docker选用的docker-ce 17.03版本；验证使用的mediaSDK和yami，驱动libva API 版本为0.39.0。

### 3.2.2 工具学习使用

在个人使用中最多使用的是MobaXterm Personal Edition工具，用于ssh连接远程服务器进行相关的操作。其经常使用功能如下：

**（1）windows下支持多标签的终端**

通过MobaXterm进行远程终端链接，你可以创建 SSH, Telnet, Rlogin, RDP, VNC, XDMCP, FTP, SFTP or 串口等链接，如图1-14所示。你的每次链接都会自动保存并且出现在左侧链接窗口中。

全能型开源远程终端：MobaXterm

图1-14 可支持的终端链接

对于本机，MobaXterm让你可以在windows下运行Unix命令，如:ls, cd, grep, awk, tail, cut, sed, wget, rsync, telnet, ssh, rlogin, rsh…等等Unix基本命令。

**（2）图形化的SFTP 浏览器**

当你链接一个ssh终端的时候，左侧窗口就会出现一个图形化的SFTP 浏览器，它可以让你通过安全的SFTP 链接拖放文件来管理远程服务器。

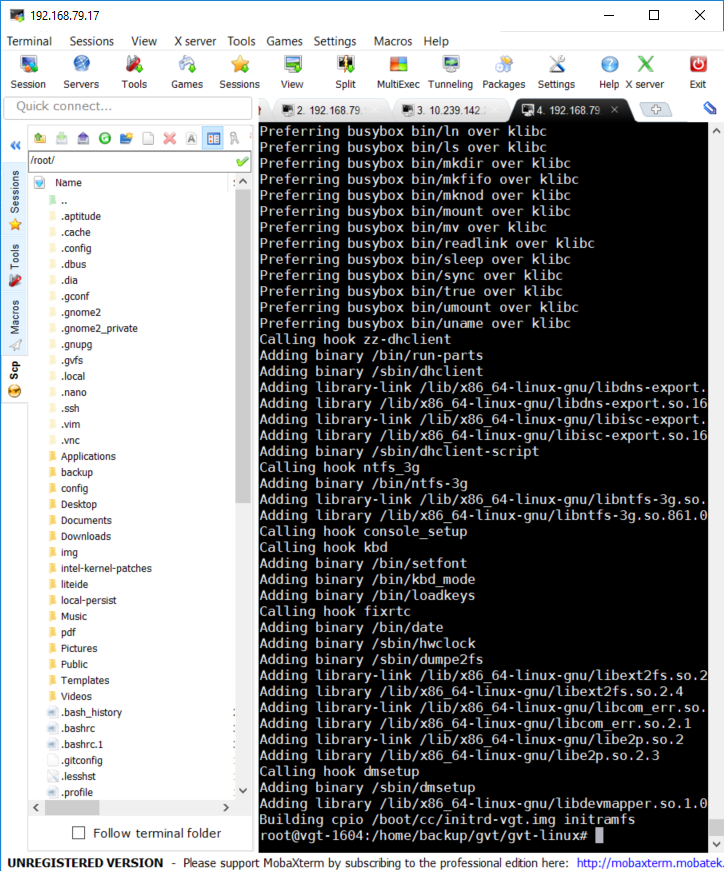


图1-15 操作界面

**（3）分屏显示及多任务执行**

MobaXterm支持多分屏显示，方便管理多台服务器，并且你可以仅输入一次，让一条命令同时在这些不同的服务器终端执行。

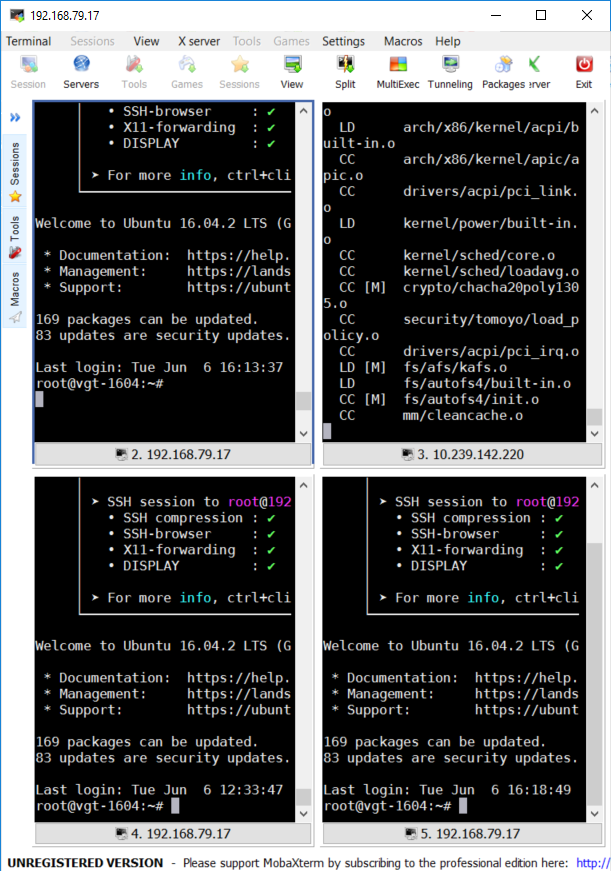


图1-16 分屏操作

### 3.2.3 其他学习内容

**（1）mediaSDK安装与配置**

当前版本的mediaSDK可以安装在CentOS7.2.1511版本或者添加了patch的linux4.4 kernel，其中CentOS上完全支持脚本自动化安装，如采用linux 4.4 kernel需要将软件包目录下的patch添加进代码中进行编译kernel，之后再安装libva等driver。

**（2）yami安装与配置**

Yami的安装需要libva-dev的支持，我之前采用的是用ubuntu16.04直接安装的0.39.0版本，后来在导师提醒下开始使用自己编译的最新版本的0.40.0版本的libva。此外还需要一个intel vaapi driver。

## 3.3. 职业素养学习培养

### 3.3.1 软件工程系统

1. 在评价复杂软件工程的解决方案的影响之前，我认为需要知道工程理论知识是能够将数学、自然科学、工程基础和专业知识用于解决复杂工程问题的专用知识集合，因此自然而然需要我们做到：
2. 掌握数理及自然科学的基础知识；
3. 掌握软件工程基础及专业知识；
4. 运用解决复杂工程问题方法。
5. 关于问题分析方面，我认为能够应用数学、自然科学和工程科学的基本原理，识别、表达、并通过文献研究分析复杂工程问题，以获得有效结论，这是我们应该遵循的一个思考过程，结合实训实习经验，我认为对于一个复杂的工程问题的分析应该遵循如下几点：
6. 运用数理方法制定分析模型；
7. 运用草稿、图表、流程表等工程方法描述问题；
8. 通过文献研究分析复杂工程问题。
9. 关于设计/开发解决方案方面，除了能够设计针对复杂工程问题的解决方案，设计满足特定需求的系统、单元模块，我觉得还能够在设计环节中体现创新意识，考虑社会、健康、安全、法律、文化以及环境等因素，具体。

① 掌握软件生命周期要素，了解软件开发过程管理模型、ISO国际质量标准、CMMI软件评价标准；

② 设计符合需求、具有创新性的解决方案的能力，同时识别社会、健康、安全、法律、文化以及环境等因素；

1. 构建符合规格及限制的单元及系统。
2. 关于预研，我觉得需要能够基于科学原理并采用科学方法对复杂工程问题进行研究，包括设计实验、分析与解释数据、并通过信息综合得到合理有效的结论，判断设计的合理性。

① 设计测试项目的能力，包括资料收集、参数值范围设定、及统计分析方法的运用能力；

② 分析与解释数据的能力，包括实施实验，并以预测或预期结果进行实验比较 ；

1. 通过信息综合得到合理有效的结论的能力，包括从实验中呈现资料、解释资料或结果，并总结结论及给予建议，形成报告。
2. 使用现代工具：能够针对复杂工程问题，开发、选择与使用恰当的技术、资源、现代工程工具和信息技术工具，包括对复杂工程问题的预测与模拟，并能够理解其局限性。
3. 了解现代工程工具和信息技术工具的使用方法；
4. 掌握开发、选择与使用现代工具；
5. 对复杂工程问题进行预测与模拟，并能够理解其局限性。

### 3.3.2 职业道德规范

作为一名软件工程师，为了实现对工程系统质量，安全和利益的承诺目标，我觉得应当坚持以下八项原则：

1. 公众----作为软件工程师应当始终如一地以符合公众利益为目标；
2. 客户和雇主----在保持与公众利益一致的原则下，我们软件工程师应满足客户和雇主的最高利益；
3. 产品----作为软件工程师应当确保我们的产品和相关的改进符合可能达到的最高专业标准；
4. 判断----作为软件工程师在进行相关的专业判断时应该坚持正直，诚实和独立的原则；
5. 管理----作为软件工程的管理和领导人员在软件开发和维护的过程中，应当自觉遵守，应用并推动合乎道德规范的管理方法；
6. 专业----作为软件工程师应当自觉推动本行业所提倡的诚实，正直的道德规范，并自觉维护本行业的声誉，使软件行业更好的为公众利益所服务；
7. 同僚----作为软件工程师对其同僚应持平等互助和支持的态度；
8. 自身----作为软件工程师应终身不断地学习和实践其专业知识，并在学习和实践的过程中不断提高自身的道德规范素养。

## 3.4. 工程协作与交流情况

本项目在公司第二季度开始实施的，每周二中午都会进行各自工作的汇总和下一步工作的重点分析。本项目需要完成kernel添加和docker修改以及plugin的分析，导师负责的kernel部分的cgroup的添加；另一个工程师负责docker部分关于cgroup的添加和一些参数的添加；我负责设计docker gpu plugin并参与部分讨论，验证一些方案可行性等等。

第一次讨论会主要关于整个项目的一个总体分析和可以采取的技术分析，我阐述了关于docker如何将device资源尤其是GPU device分配给container的过程分析；我们主要讨论了如下几点:

1. 研究如何给docker添加新的cgroup支持，是否需要通过插件或者修改docker server端代码；
2. 由于mediaSDK测试需要特定的kernel，所以讨论了将测试软件换成yami；
3. Docker 源码编译中的一些问题，比如需要设置代理等。

第一次讨论的核心就是怎么添加cgroup，这也成为我们后期工作的主线。第二次我觉得比较重要的讨论的主要内容如下所示：

1.完成Linux kernel 中GPU priority的cgroup的添加；

2.跑通docker源码编译的流程;

3.研究nvidia-docker的实现形式;

4.接下来的小组工作重心: 审计GFX memory，给libcontainer添加gpu cgroup支持，测试clear container性能等。

后面的讨论主要围绕如下几个方面展开的：

1. 容器GPU lib兼容性验证工具开发；
2. 设计新的驱动版本检查方法；
3. 检查docker在clear container中的使用方法；
4. Yami在clear container中的安装和配置；
5. 尝试移植gpu\_cgroup给4.4kernel，解决mediaSDK的使用；
6. 更新clear container启动的wiki。

通过不断的汇总和讨论，推动了整个项目的有序进行。

# 4. 存在问题与解决方案

## 4.1. 存在的主要问题

### 4.1.1 问题总结

1.初期对于docker-ce cgroup的控制流程了解得不完善;

2.对于利用plugin进行版本检查的可能性判断不清楚;

3.docker源码编译中遇到的一些问题。

### 4.1.2 解决方式

对于docker-ce cgroup流程问题采用runc来替代分析，runc也是一个可以用于管理容器的部分，里面对于cgroup的代码流程相对要清晰许多；docker-ce的本质就是runc，libcontainer，daemon，client等的组合产品，所以从局部模块入手比整体更利于开发。

对于plugin 检查driver版本的问题，因为plugin的类别是docker daemon规定好的，而采用volume plugin的话，如果要在创建容器之前检查起内部driver版本，则需要通过比如docker volume inspect方式获取其内部的驱动信息，而问题在于比如mediaSDK这种软件好像并没有显示的版本信息，只有通过运行小的测试来判断匹配与否。什么时候运行版本检测脚本？什么方式运行？这些仍然是我们面临的很大的疑惑和困难。我们通过分析nvidia docker的实现，也开始考虑使用版本标签的形式来替代这样一个插件动态检测的行为，具体实现还值得深究。

在docker源码编译中遇见的问题有第一次编译过程中需要很长的时间，特别是在下载MacOSX10.11.sdk的环节，大概需要30分钟左右，期间没有任何输出，但是确实还在正常进行；第二个是当你在make build时出现错误后可以再次执行一次，在编辑Dockerfile的时候也是这样处理，缓存的结果在再次编译的时候将会用上；此外，需要在Dockerfile中删除如下的一句，否则会造成没有权限的错误。

|  |
| --- |
| buildpack-deps:jessie@sha256:85b379ec16065e4fe4127eb1c5fb1bcc03c559bd36dbb2e22ff496de55925fa6 \ |

在实际过程中可能会遇到网络方面的问题，一是需要配置公司的代理，二是需要修改/etc/resolve.conf的DNS解析。Docker如果采用overlay2的话，目前是不支持nfs文件系统，所以这一点需要注意。Docker的编译由于依赖极其复杂，所以需要先根据官方wiki安装docker，并创建一个container作为编译的环境，在这个环境里面才能开始修改后的docker源码编译。

### 4.1.3 收获

通过本次项目，对kernel cgroup有了更加深入的理解，同时对docker和docker plugin的操作和实现原理有了一些了解，也收获了团队协作的精神品质和认真负责的职业操守。

## 4.2. 解决方案与可行性研究

### 4.2.1 尚未解决的问题

设计方案，用于在创建容器之前进行驱动版本检查，以避免因驱动版本问题造成的影响。

### 4.2.2 解决方案

解决方案一：静态检测法，获取driver lable进行直接比较；

解决方案二：动态检测法，通过运行标准测试用例，根据结果进行判断；

解决方案三：后台隐藏法，第一个起的container处于后台运行模式，在该模式下运行预置的脚本和测试用例，如果测试不通过，则执行退出container操作并删除container，显示相关driver错误；

解决方案四：插件+动态检测法，鉴于目前docker的支持的插件类别有一定的要求，所以这个方案还需要进一步实验，比如plugin执行命令以及挂载容器的filesystem方面。

### 4.2.3 可行性分析

考虑到Docker plugin的局限性和特殊，以及结合判断动态检测法的可操作性，我们加以分析nvidia同类产品的方案来进行可行性分析。

Nvidia针对Docker Container推出了自己的解决方案—nvidia docker + nvidia docker plugin的形式，nvidia docker是在原生docker上封装了一层，对docker命令行接口进行一些替换和处理，如create和run命令；nvidia docker plugin则是充当一个daemon进程，用于发现host的驱动文件和GPU硬件，并对docker daemon发起的存储挂载请求做出响应。下面是关于nvidia docker对于驱动版本检查机制的分析，整体分析如图2-1所示。

1. **整体架构分析**



图2-1 nvidia docker整体架构

Nvidia docker作为在原生docker之上的一层，对于create和run参数有着其自定义的处理步骤，之后才将处理结果传给原生docker进行处理。Nvidia docker与原生的docker交互发生在cudaSupported函数，其具体实现如图2-3所示。在nvidia docker获取image驱动版本的实现中考虑了remote和local两种情况，remote模式下采用json格式的文本传输获取CUDA版本；local模式下采用GetCUDAVersion函数实现CUDA版本的获取，该函数具体分析如图2-2所示。

1. **获取CUDA版本**



图2-2 local模式获取cuda版本

在local模式下获取cuda版本，比较特殊的一步是driverGetVersion的本质是采用了go语言调用c语言是形式，引入cuda\_runtime\_api头文件，用go语言调用其cuda api里面的cudaDriverGetVersion函数来获取cuda版本。这里所获取到的版本将会用于和容器image里的lable版本进行比较得出驱动是否匹配的信息，以此决定是否创建一个容器。与Image里面的lable所进行比较的是该驱动所能支持的最大的CUDA版本。其具体流程如图2-3所示。

1. **获取lable版本**



图2-3 获取image版本标签进行匹配

在编译CUDA image的Dockerfile里我们可以看见关于com.nvidia.cuda.

-version的定义，通过Lable可以指定image的版本，说明等；nvidia docker本质上还是调用了原生的docker的inspect命令，来获取image里面所定义的lable信息，docker inspect –-format 命令可以用于获取image或container里面的元数据，即这里所需要的com.nvidia.cuda.version标签信息。

### 4.2.4 结论

结合docker plugin实现方式和nvidia实现方案分析，采用docker plugin进行动态检测的话，需要解决如何在创建容器之前检查完容器内部的驱动版本，docker inspect可以获取image里的信息，所以静态信息检测法可以；而如果要在宿主机上用容器所含驱动库来执行一个测试脚本，从而根据运行结果判断驱动是否匹配，这个可以采取volume plugin将容器filesystem挂载到宿主机上。

# 5. 前期任务完成度与后续实施计划

## 5.1 前期任务完成度

对于前期的导师交给的任务都基本完成了，除了个别因为系统环境过于单薄，安装过程不详造成的问题，但这些均通过替换方案的方式得到了部分或间接的解决。初期完成了Docker container GPU cgroup方案的可行性研究，在多个Docker container中安装配置mediaSDK采集相应的数据进行分析，并编写内部wiki记录整个步骤；尝试编译docker 源代码，根据docker sdk 用go语言编写插件，并监听其端口进行分析研究；分析nvidia docker实现机制，为导师提供相关分析信息；创建基于GVT-g技术的Clear Container，在里面部署yami进行性能分析，编写相关wiki记录整个步骤；在Clear Container里面进行docker in container方案设计和验证，编写wiki记录整个步骤。

## 5.2 后续实施计划

### 5.2.1 实施方案计划

后续将着重研究采用插件式检查驱动版本和采用数据标签的方案，根据目前的信息，Docker存储插件更多的表现出来的是在数据存储和共享方面的功能，至于如何让其通过运行某些标准测试用例来判断驱动版本还值得深思。特别是检查操作是在创建一个container之前需要完成的话，那么怎么将镜像里的驱动库挂载到宿主机上，并在其环境中执行相应的检测脚本。目前可以先验证plugin自动执行命令的可能性，在此基础上结合volume plugin或者docker inspect将容器filesystem挂载到宿主机环境执行检测脚本。

### 5.2.2 知识技能学习计划

剩下时间还需要加强linux内核学习，特别是namespace和cgroup学习；还需要复习一下操作系统的知识；为了加强对计算机体系结构的认识，需要将自己的《深入理解计算机系统》一书好好再研究。此外，对于比如git版本控制目前还只会基本操作，对于大型工程维护和更新还需要更深入学习和了解。

# 参考文献

1. <https://docs.docker.com/engine/installation/linux/ubuntu/>，Docker安装wiki
2. <https://github.com/01org/libyami/wiki/Build> ，Yami安装wiki
3. <https://github.com/NVIDIA/nvidia-docker/wiki> , nvidia-docker 官方文档
4. Kun Tian,Yaozu Dong,David Cowperthwaite,A Full Virtualization Solution with Mediated Pass-Through[J],Usenix Conference on Usenix Technical Conference,2014:121-132
5. 系统虚拟化：原理与实现/英特尔开源软件技术中心，复旦大学并行处理研究所著.—北京：清华大学出版社，2009.3