KYLINSOFT
麒麟软件

银河麒麟云底座操作系统 V10

系统管理员手册

麒麟软件有限公司

2024 年 7 月

商标声明

KYLINSOFT 麒麟软件和其他麒麟商标均为麒麟软件有限公司的商标。本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受麒麟软件有限公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，麒麟软件有限公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容有可能变更，麒麟软件有限公司保留在没有任何通知或提示的情况下对内容进行修改的权利。除非另有约定，本文档仅作为使用指导，并不确保手册内容完全没有错误。本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

目 录

银河麒麟云底座操作系统最终用户许可协议

本最终用户使用许可协议（以下简称“本协议”）是贵方（作为实体或个人）与麒麟软件有限公司（以下简称“许可证颁发者”）之间的法律协议。本协议适用于由麒麟软件有限公司开发并制作发行的银河麒麟云底座操作系统软件产品，（以下简称“本软件”）

请贵方认真阅读本协议，一旦购买、安装、下载或以其它方式使用本软件（包括其组件），即表示贵方同意本协议的条款；如果贵方不同意这些条款，则不得下载、安装或使用本软件，贵方应该通知本软件的卖方以获得退款。代表某实体行事的个人表示其有权代表该实体签署本协议。

1. 定义

使用：指的是下载、安装、复制、运行、展示或以其他方式利用银河麒麟云底座操作系统软件产品的行为。

用户：指的是使用或安装本软件的个人、团体、公司或组织。

组织：指一个法人实体，不包括为税收或法律人格目的而单独存在的子公司和分公司。例如，私营领域的组织可以是一个有限公司、合伙企业或联合企业，但不包括该组织旗下具有单独的报税识别号码或公司注册号码的子公司或分公司；公营领域的组织可以是一个特定的政府机构或当地政府公共机构。

2. 使用许可

本软件及其各个组件都归许可证颁发者或其他许可证颁发者所有，并受著作权法和其他相关法律的保护。在遵守本协议条款和条件的前提下，许可证颁发者授予贵方永久、不可转让、全球范围内的著作权普通许可，除符合本协议第 5 条再分发条件以外，仅允许在贵方组织（定义如上）内部重制和使用本软件的副本。

3. 第三方/开放源代码

对于本软件中包含的任何开放源代码，本协议的任何条款均不得限制、约束或以其他方式影响任何适用开放源代码许可证所赋予贵方的任何相应的权利或义务或贵方应遵守的各种条件。本软件可能包含或捆绑有其他软件程序，这些软件程序使用不同的条款许可，并/或由许可证颁发者之外的第三方许可。使用附带单独许可协议的任何软件

程序需受该单独许可协议的约束。

4. 订阅服务

除非贵方购买的订阅产品中明确包含支持维护或支持，否则许可证颁发者没有义务提供此类服务。许可证颁发者会销售本软件的订阅产品，使贵方能够付费获得在指定年周期提供的技术支持和/或软件更新的内部使用权（以下简称“订阅产品”），这些订阅产品受适用的《银河麒麟云底座操作系统订阅协议》的各项条款的约束。

5. 商标和标识

贵机构承认并与麒麟软件有着以下共识，即麒麟软件拥有麒麟软件、银河麒麟商标，以及所有与麒麟软件、银河麒麟相关的商标、服务标记、标识及其他品牌标识（以上统称为“麒麟软件标记”）。贵机构对麒麟软件标记的任何使用都应有利于麒麟软件。

只有在以下情况下方可对本软件进行商业性质的再分发：(a) 得到许可证颁发者通过独立的书面协议对该类商业再分配授予的许可，(b) 贵方去除和替换了所有出现的任何麒麟软件标记。

6. 许可限制

本软件及其各个组件均归许可证颁发者和/或其他许可证颁发者所有，并受著作权法和其他相关法律的保护。根据适用的许可证，对本软件及其任何组件或其任何复制、修改或合并部分的所有权属上述权利人所有。许可证颁发者保留所有未明确授予贵方的权利。除符合本协议第 5 条再分发条件以外，本软件仅许可贵方内部使用。

7. 所有权

本软件的所有权并未转让给贵方。许可证颁发者和/或其第三方许可证颁发者保留本软件和服务（包括本软件的任何改编版本或副本）中所有知识产权的全部权利、所有权和利益。本软件并非出售给贵方，贵方获得的只是使用本软件的有条件许可证。通过本软件访问的内容的相关权利、所有权和知识产权是相应内容所有者的财产，并可能受相应的著作权法或其他相关法律的保护。本协议未授予贵方对此类内容的任何权利。

8. 有限担保

麒麟软件向贵方担保，自购买或其它合法取得之日起九十（90）天内（以收据副本为凭证），本软件的存储介质（如果有的话）在正常使用的情况下无材料和工艺方面的

缺陷。除上述内容外，本软件按“原样”提供。在本有限担保项下，贵方的所有补偿及麒麟软件的全部责任为由麒麟软件选择更换本软件介质或退还本软件的购买费用。

9. 审查

麒麟软件的代表或其指定人员有权基于《银河麒麟云底座操作系统订阅协议》中的订阅条款和条件，核实贵方是否遵守本协议。贵方同意：(a)及时回应索要信息、文件和/或记录的请求；(b)就现场访问授予适当的出入权限，以确认贵方的合规性；以及(c)合理地配合任何该等核实工作。麒麟软件将至少提前十(10)天以书面形式通知任何现场访问，并将在正常工作时间内进行现场访问，以合理的方式尽可能减少对贵方业务的干扰。如果麒麟软件通知贵方有任何不合规或付款不足的情况，贵方应在通知之日起十五(15)天内解决该等不合规和/或付款不足的问题。如果付款不足超过百分之五(5%)，贵方还得向麒麟软件偿付检查费用。

10. 担保的免责声明

除非在本协议中有明确规定，否则对于任何明示或默示的条件、陈述及担保，包括对适销性、对特定用途的适用性或非侵权性的任何默示的担保，均不予负责，但上述免责声明被认定为法律上无效的情况除外。

11. 责任限制

在法律允许范围内，无论在何种情况下，无论采用何种有关责任的理论，无论因何种方式导致，对于因使用或无法使用本软件引起的或与之相关的任何收益损失、利润或数据损失，或者对于特殊的、间接的、后果性的、偶发的或惩罚性的损害赔偿，麒麟软件或其许可方均不承担任何责任（即使麒麟软件已被告知可能出现上述损害赔偿）。根据本协议，在任何情况下，无论是在合同、侵权行为（包括过失）方面，还是在其他方面，麒麟软件对贵方的责任将不超过贵方就本软件所支付的金额。即使上述担保未能达到其基本目的，上文所述的限制仍然适用。

12. 终止

本协议在终止之前有效。贵方可以随时终止本协议，但必须同时销毁本软件的全部正本和副本。如果贵方未遵守本协议的任何规定，则本协议将不经麒麟软件发出通知立即终止。终止时，贵方必须销毁本软件的全部正本和副本，并且需承担因未遵守本协议

而导致的法律责任。

13. 管辖法律适用

与本协议相关的任何争议解决（包括但不限于诉讼、仲裁等）均受适用中华人民共和国法律管辖。任何其它国家和地区的选择法律规则不予适用。

14. 可分割性

如果本协议中有任何规定被认定为无法执行，则删除相应规定，本协议仍然有效，除非该删除会防碍各方愿望根本目的的实现（在这种情况下，本协议将立即终止）。

15. 完整性

本协议是贵方与麒麟软件就其标的达成的完整协议。它取代此前或同期的所有和本协议不一致的口头或书面往来信息、建议、陈述和担保。在本协议期间，有关报价、订单、回执或各方之间就本协议标的进行的其他往来通信中的任何冲突条款或附加条款，均以本协议为准。对本协议的任何修改均无约束力，除非通过书面进行修改并由每一方的授权代表签字。

16. 因侵权而终止

如果本软件成为或在任一方看来可能成为任何知识产权侵权索赔之标的，则任一方可立即终止本协议。

17. 其他条款

协议提供中英文两种版本，以上任何内容如有歧义，以中文版本为准。

1. 银河麒麟操作系统隐私政策声明

版本发布日期：2023 年 09 月 20 日

版本生效日期：2023 年 09 月 20 日

尊敬的银河麒麟操作系统用户（以下简称“您”），银河麒麟操作系统系列软件产品是由麒麟软件有限公司（以下简称“我们”或“麒麟软件”）研制发行的，用于办公或构建企业及政府的信息化基础设施。

麒麟软件非常重视您的个人信息和隐私保护，在您使用本产品的过程中，我们会按照《银河麒麟操作系统隐私政策声明》（以下简称“本声明”）收集、存储、使用您的个人信息。为了保证对您的个人隐私信息合法、合理、适度的收集、使用，并在安全、可控的情况下进行传输、存储，我们制定了本声明。我们将向您说明收集、保存和使用您的个人信息的方式，以及您访问、更正、删除和保护这些信息的方式。我们将会按照法律要求和业界成熟安全标准，为您的个人信息提供相应的安全保护措施。如您点击或勾选“同意”并确认提交，即视为您同意本隐私政策声明，并同意我公司将按照本政策来收集、存储和使用您的相关信息。

本声明将帮助您了解以下内容：

一、关于收集和使用涉及您的个人信息

二、如何存储和保护涉及您的个人信息

三、如何管理您的个人信息

四、关于第三方软件的隐私说明

五、关于未成年人使用产品

六、本声明如何更新

七、如何联系我们

一、如何收集和使用您的个人信息

1.收集涉及您的个人信息的情况

我们在您使用银河麒麟操作系统产品过程中收集相关的信息，主要为了向您提供更高质量、更易用的产品和更好的服务。

1）银河麒麟操作系统的产品授权许可机制，会根据您所使用计算机的网卡、固件和主板等信息通过加密机制和转换方法生成申请产品正式授权许可的机器码；您将该机

器码发给麒麟软件商务人员根据合同及相关协议可申请正式许可。该机器码不包含您所使用计算机的网卡、固件和主板等设备具体信息。

2）银河麒麟操作系统应用商店的服务器端，会根据您所使用计算机的 CPU 类型信息以及 IP 地址进行连接；实现您方便快捷使用应用商店。您所使用计算机的 IP 地址可能会记录在应用商店的服务器端系统的日志中。

3）银河麒麟操作系统的升级更新，会根据您所使用计算机的 IP 地址进行连接；以便实现您确认是否更新升级系统。

4）使用银河麒麟操作系统产品过程中，因业务往来及技术服务等您提供的电子邮箱、电话、姓名等个人信息。

5）银河麒麟操作系统可能提供生物识别相关功能，会存储身份鉴别相关的信息在您的机器。这部分信息我们不收集和上传服务器。

以后银河麒麟操作系统产品升级过程中，如新增涉及个人信息收集部分，将及时更新本部分内容。

2. 使用涉及您的个人信息的情况

我们严格遵守法律法规的规定及与用户的约定，将收集的信息用于以下用途。若我们超出以下用途使用您的信息，我们将再次向您进行说明，并征得您的同意。我们会将收集的信息用于以下用途：

产品功能：主要涉及产品许可机制、应用商店使用、系统更新维护、生物识别等需要。

安全保障：为保障您使用银河麒麟操作系统的安全，我们会利用相关信息协助提升产品的安全性、可靠性和可持续服务。

与您沟通：我们会利用收集的信息（例如您提供的电子邮件地址、电话等）直接与您沟通。例如，业务联系、技术支持或服务回访。

产品改进：将收集的信息用于改进产品当前的易用性、缺陷以及提升产品用户体验等。

为了遵从相关法律法规、部门规章、政府指令的相关要求。

3.信息的分享及对外提供

我们不会共享或转让您的个人信息至第三方，但以下情况除外：

1）获取您的明确同意：经您事先同意，我们可能与第三方分享您的个人信息；

2）为实现外部处理的目的，我们可能会与关联公司或其他第三方合作伙伴（第三方服务供应商、承包商、代理、应用开发者等）分享您的个人信息，让他们按照我们的说明、隐私政策以及其他相关的保密和安全措施来为我们处理上述信息，并用于向您提

供我们的服务，实现“如何收集和使用您的个人信息”部分所述目的。如我们与上述关联公司或第三方分享您的信息，我们将会采用加密、匿名化处理等手段保障您的信息安全。

3）我们不会对外公开披露所收集的个人信息，如必须公开披露时，我们会向您告知此次公开披露的目的、披露信息的类型及可能涉及的敏感信息，并征得您的明示同意。

4）随着我们业务的持续发展，我们有可能进行合并、收购、资产转让等交易，我们将告知相关情形，按照法律法规及不低于本声明所要求的标准继续保护或要求新的控制者继续保护您的个人信息。

5）我们可能基于法律要求或相关部门的执法要求披露您的个人信息。

如我们使用您的个人信息，超出了与收集时所声称的目的及具有直接或合理关联的范围，我们将在使用您的个人信息前，再次向您告知并征得您的明示同意。

根据相关法律法规以及国家标准，在以下情况下我们可能会收集、使用您的个人信息，征得授权同意的例外情况：

1）与国家安全、国防安全等国家利益直接相关的；

2）与公共安全、公共卫生、公众知情等重大公共利益直接相关的；

3）与犯罪侦查、起诉、审判和判决执行等直接相关的；

4）出于维护您或其他个人的生命、财产等重大合法权益但又很难得到您本人同意的；

5）所收集的个人信息是您自行向社会公众公开的；

6）从合法公开披露的信息中收集的个人信息，如合法的新闻报道、政府信息公开等渠道；

7）根据您要求签订和履行合同所必需的；

8）用于维护所提供的产品或服务的安全稳定运行所必需的。如发现、处置产品或服务的故障；

9）出于公共利益开展统计或学术研究所必需，且其对外提供学术研究或描述的结果时，对结果中所包含的个人信息进行去标识化处理的；

10）法律法规规定的其他情形。

二、我们如何存储和保护涉及您的个人信息

1.信息存储的地点

我们会按照法律法规规定，将在中国境内收集和产生的个人信息存储于中国境内。

2.信息存储的期限

一般而言，我们仅为实现目的所必需的时间保留您的个人信息。记录在日志中的信息会按配置在一定期限保存及自动删除。

当我们的产品或服务发生停止运营的情形时，我们将以通知、公告等形式通知您，在合理的期限内删除您的个人信息或进行匿名化处理，并立即停止收集个人信息的活动。

3.我们如何保护这些信息

我们努力为用户的信息安全提供保障，以防止信息的丢失、不当使用、未经授权访问或披露。

我们将在合理的安全水平内使用各种安全保护措施以保障信息的安全。例如，我们会使用加密技术（例如，SSL/TLS）、匿名化处理等手段来保护您的个人信息。

我们建立专门的管理制度、流程和组织以保障信息的安全。例如，我们严格限制访问信息的人员范围，要求他们遵守保密义务，并进行审计。

4.若发生个人信息泄露等安全事件，我们会依法启动应急预案，阻止安全事件扩大，并以推送通知、公告等形式告知您安全事件的情况、事件可能对您的影响以及我们将采取的补救措施。我们还将按照法律法规和监管部门要求，上报个人信息安全事件的处置情况。

三、如何管理您的个人信息

如果担心因使用银河麒麟操作系统产品导致个人信息的泄露，您可根据个人及业务需要考虑暂停或不使用涉及个人信息的相关功能，如产品正式授权许可、应用商店、系统更新升级、生物识别等。

在使用银河麒麟操作系统之上使用第三方软件时，请注意个人隐私保护。

四、关于第三方软件的隐私说明

您在使用银河麒麟操作系统之上安装或使用第三方软件时，第三方软件的隐私保护和法律责任由第三方软件自行负责。

您在使用银河麒麟操作系统之上安装或使用第三方软件时，请您仔细阅读和审查对应的隐私声明或条款；注意个人隐私保护。

五、关于未成年人使用产品

银河麒麟操作系统系列产品仅供成年人使用，如果您是未成年人，则需要您的监护人同意您使用本产品并同意相关服务条款。父母和监护人也应采取适当的预防措施保护未成年人，包括监督其对银河麒麟操作系统系列产品的使用。

六、本声明如何更新

我们保留适时更新本声明的权利，当本声明发生变更时，我们会通过产品安装过程或公司网站向您展示变更后的声明，只有在获取您的同意后，我们才会按照更新后的声明收集、使用、存储您的个人信息。

七、如何联系我们

如您对本声明存在任何疑问，或任何相关的投诉、意见，请联系麒麟软件客服热线400-089-1870、官方网站（www.kylinos.cn）以及麒麟软件进行咨询或反映。您可以通过发送邮件至 market@kylinos.cn 方式与我们联系。

受理您的问题后，我们会及时、妥善处理。一般情况下，我公司将在 15 个工作日内给予答复。

本声明自更新之日起生效，同时提供中英文两种版本，以上任何条款如有歧义，以中文版本为准。

最近更新日期：2019 年 12 月 18 日

地址：天津市滨海高新区塘沽海洋科技园信安创业广场 3 号楼（300450）

北京市海淀区北四环西路 9 号银谷大厦 20 层（100190）

长沙市开福区三一大道 156 号工美大厦 10 楼（410073）

电话：天津（022）58955650 北京（010）51659955 长沙（0731）88280170

传真：天津（022）58955651 北京（010）62800607 长沙（0731）88280166

公司网站：www.kylinos.cn

电子邮件：support@kylinos.cn

特别提示说明

银河麒麟云底座操作系统 V10 同源支持 intel、飞腾、海光、鲲鹏等自主 CPU 平台。本手册主要面向系统管理员及相关技术人员，如本手册未能详细描述之处，有需要请致电麒麟软件有限公司技术服务部门。

 重要：

本手册中命令、操作步骤等举例仅供参考，命令执行的输出信息等在不同 CPU 平台或因操作系统或组件的版本升级可能有少许差异；本手册尽量加以说明。如有差异之处，请以银河麒麟云底座操作系统 V10 在具体 CPU 平台上实际操作或输出信息为准。

1. 基本系统配置

1.1. 系统地区和键盘配置

系统地区配置是指系统服务和用户界面的语言环境配置。键盘布局配置是指文本控制台和图形用户界面的键盘布局规则。这些设置可以通过修改/etc/locale.conf 配置文件或使用 localectl 命令。此外，您可以在用户图形界面来执行任务，详情请参考安装手册。

1.1.1. 配置系统地区

系统地区配置文件为/etc/locale.conf，在系统启动时引导 systemd 守护进程。这个配置文件可以被每一个服务或者用户继承，单个服务或者用户也可修改配置文件。例如语言为英语，地区为德国的/etc/locale 文件的配置内容如下：

```
LANG=de_DE.UTF-8
LC_MESSAGES=C
```

LC_MESSAGES 选项决定了诊断消息的标准输出文本格式。其他选项说明总结在下表。

配置项	描述
LANG	提供系统时区的默认值
LC_COLLATE	定义该环境的排序和比较规则
LC_CTYPE	用于字符分类和字符串处理，控制所有字符的处理方式，包括字符编码，字符是单字节还是多字节，如何打印等。是最重要的一个环境变量
LC_NUMERIC	非货币的数字显示格式
LC_TIME	时间和日期格式
LC_MESSAGES	提示信息的语言

1.1.1.1. 显示当前配置

Localectl 命令可用于配置语言环境和键盘布局。显示当前配置，可使用如下命令：

```
#localectl status
```

1.1.1.2. 显示可用地区列表

显示可用地区列表可使用如下命令：

```
#localectl list-locales | grep -E 'en|zh'
```

1.1.1.3. 配置地区

配置系统默认地区，需要以 root 用户身份运行：

```
#localectl set-locale LANG=locale
```

用户可以配置适合的地区标示符以代替 locale，可通过 localectl list-locales 检索适合的地区。

1.1.2. 配置键盘布局

键盘布局配置是指文本控制台和图形用户界面的键盘布局规则。

1.1.2.1. 显示当前配置

Localectl 命令可用于配置语言环境和键盘布局。显示当前配置，可使用如下命令：

```
#localectl status
```

1.1.2.2. 显示可用键盘布局列表

显示可用键盘布局列表可使用如下命令：

```
#localectl list-keymaps
```

1.1.2.3. 配置键盘

配置系统默认键盘布局，需要以 root 用户身份运行：

```
#localectl set-keymap {map}
```

用户可以配置适合的键盘布局标示符以代替{map},如“cz”，可通过 localectl list-keymaps 检索适合的键盘布局。该命令还可用于配置 X11 窗口的键盘布局映射，但使用--no-convert 参数的话则不生效。同样也可用以下命令单独配置 X11 窗口的键盘布局：

```
#localectl set-x11-keymap cz
```

如果用户希望 X11 窗口和命令行终端的键盘布局不一样，可以使用如下命令：

```
#localectl --no-convert set-x11-keymap cz
```

1.1.3. 其他资源

其他官方配置系统地区和键盘布局的内容可以参考安装手册。同时还可参考 1.6 获取特权章节。

1.2. 网络访问配置

1.2.1. 动态网络配置

打开终端，以网口 eth0 为例：

```
#nmcli conn add connection.id eth0-dhcp type ether ifname eth0 ipv4.method auto
```

其中“eth0-dhcp”为连接的名字，可以根据自己的需要命名方便记忆和操作的名字；“ifname eth0”为配置的网口，根据自己的设备情况按需调整。

1.2.2. 静态网络配置

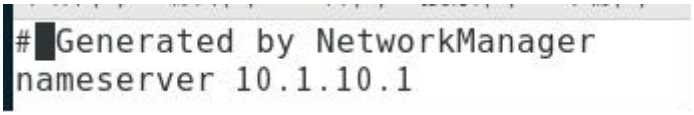
打开终端，以网口 eth0 为例：打开终端，编辑/etc/resolv.conf，设置 nameserver：

```
#nmcli conn add connection.id eth0-static type ether ifname eth0 ipv4.method manual ipv4.address 192.168.1.10/24 ipv4.gateway 192.168.1.254 ipv4.dns 192.168.1.254
```

其中“eth0-static”为连接的名字，可以根据自己的需要命名方便记忆和操作的名字；“ifname eth0”为配置的网口，根据设备情况按需调整；IP、子网掩码、网关根据实际网络按需配置。

1.2.3. 配置 DNS

打开终端，编辑/etc/resolv.conf，设置 nameserver：



1.3. 日期和时间配置

操作系统区分以下两种时区：

- 实时时间（RTC），通常作为物理时钟，它可以独立于系统当前状态计时，在主机关机情况下也可计时。
- 系统时间，是基于实时时间的由操作系统内核维护的软件时间。等系统启动内核初始化系统时间后，系统时间就独立于实时时间自行计时。

系统时间通常还保持一套世界统一时间（UTC），用于转换系统的不同时区，本地时间就是用户所在时区的真实时间。

操作系统提供了三种命令行时间管理工具，timedatectl、date 和 hwclock。以下将分别介绍各个工具的使用。

1.3.1. timedatectl 工具使用说明

1.3.1.1. 显示当前日期和时间

命令 timedatectl 可以显示当前系统时间和机器的物理时间及其详细信息。如下示

例是未启用 NTP 时钟同步的系统时间：

```
#timedatectl
```

变更 chrony 或 ntpd 服务状态不会主动通知 timedatectl 工具，如果想要更新服务的配置信息，请执行以下命令：

```
#systemctl restart systemd-timedated.service
```

1.3.1.1. 变更当前时间

系统默认启用时间同步服务，时间将自动同步远程时间服务器。变更当前时间前，需要以 root 用户运行以下命令取消自动同步：

```
#timedatectl set-ntp no
```

以 root 用户运行以下命令可以修改当前时间：

```
#timedatectl set-time HH: MM: SS
```

其中 HH 代表小时，MM 代表分钟，SS 代表秒数，均需两位表示。这个命令同样可以更新系统时间和物理时间，效果类似于 date --set 和 hwclock --systohc 命令。

系统默认时间配置基于 UTC，如果想基于本地时间来配置系统时间，需要以 root 用户运行以下命令修改。

```
#timedatectl set-local-rtc boolean
```

如果基于本地时间，需要将 boolean 配置为 yes（或者 y，true，t 或者 1）。如果使用 UTC 时间，则要将 boolean 配置为 no（或者 n，false，f 或者 0）。系统默认 boolean 为 no。

1.3.1.2. 变更当前日期

以 root 用户运行以下命令可以修改当前日期：

```
#timedatectl set-time YYYY-MM-DD
```

其中 YYYY 代表年份，需 4 位数表示；MM 代表月份，需两位数表示；DD 代表

日期，需两位表示。如果还需要配置时间，可以补充上时间参数，示例如下：

```
#timedatectl set-time '2023-09-12 12:00:00'
```

1.3.1.3. 修改时区

执行以下命令可以显示当前时区：

```
#timedatectl show
```

以 root 用户运行以下命令可以修改当前时区，如修改为“上海”：

```
#timedatectl set-timezone Asia/Shanghai
```

显示所有时区命令如下：

```
#timedatectl list-timezones
```

1.3.1.4. 同步系统与远程服务器时间

以 root 用户运行以下命令可以启用/禁用时间同步服务：

```
#timedatectl set-ntp boolean
```

启用与禁用需要配置 boolean 值为 yes 或者 no。例如需要自动同步一个远程时间服务器，可以执行以下命令：

```
#timedatectl set-ntp yes
```

1.3.2. date 工具使用说明

1.3.2.1. 显示当前日期和时间

命令 date 可以显示当前系统时间、时区、日期等信息。并可以通过参数--utc 显示当前时区时间。通过“format”标示符来输出特定状态。常用的 format 说明如下：

参数	描述
%H	以 HH 格式输出当前小时
%M	以 MM 格式输出当前分钟
%S	以 SS 格式输出当前秒数

%d	以 DD 格式输出当前日期
%m	以 MM 格式输出当前月份
%Y	以 YYYY 格式输出当前年份
%Z	显示时区制式，例如 CEST
%F	以 YYYY-MM-DD 格式输出当前年月日，等价于参数%Y-%m-%d
%T	以 HH:MM:SS 格式输出当前时间，等价于参数%H:%M:%S

示例如下：

```
# date
2023 年 09 月 12 日 星期二 17:06:32 CST
# date --utc
2023 年 09 月 12 日 星期二 09:06:52 UTC
# date +"%Y-%m-%d %H:%M"
2023-09-12 17:07
```

1.3.2.2. 变更当前时间

以 root 用户运行以下命令可以修改当前时间：

```
#date --set HH: MM: SS
```

其中 HH 代表小时，MM 代表分钟，SS 代表秒数，均需两位表示。这个命令同样可以更新系统时间和物理时间，效果类似于 hwclock --systohc 命令。

系统默认时间配置基于本地时间，如果想基于 UTC 时间来配置系统时间，需要以 root 用户运行以下命令修改。

```
#date --set HH: MM: SS --utc
```

1.3.2.3. 变更当前日期

以 root 用户运行以下命令可以修改当前日期：

```
#date --set YYYY-MM-DD
```

其中 YYYY 代表年份，需 4 位数表示；MM 代表月份，需两位数表示；DD 代表日期，需两位表示。如果还需要配置时间，可以补充上时间参数，示例如下：

```
# date --set '2023-09-12 12:00:00'
```

1.3.3. hwclock 工具使用说明

1.3.3.1. 显示当前日期和时间

命令 hwclock 可以显示当前系统时间、时区、日期等信息。并可以通过参数--utc 或--localtime 显示当前 UTC 时区时间和本地时间。示例如下：

```
#hwclock
2023-09-12 17:02:23.175782+08:00
```

1.3.3.2. 变更当前日期和时间

以 root 用户运行以下命令可以修改当前时间：

```
#hwclock --set --date "dd mmm yyyy HH:MM"
```

其中 dd 代表日期 HH 代表小时，MM 代表分钟，SS 代表秒数，均需两位表示。Mmm 代表月份，以月份英文三位字母简写表示，yyyy 代表年份，以四位数字表示。这个命令通过参数--utc 或—localtime 区分配置当前 UTC 时区时间和本地时间

基于 UTC 时间来配置系统时间，需要以 root 用户运行以下命令修改，示例如下。

```
#hwclock --set --date "12 Sep 2023 12:00" --utc
```

1.3.3.3. 同步系统与远程服务器时间

以 root 用户运行以下命令同步远程时间：

```
#hwclock --systohc
```

1.4. 用户配置

可以对用户进行创建与删除，使用 useradd 命令新建用户，使用 passwd 命令设置用户密码，使用 userdel 命令删除用户。

例如，以下命令将创建一个名为 newuser 的用户：

```
#sudo useradd newuser
```

使用 passwd 命令为 newuser 用户设置密码：

```
#sudo passwd newuser
```

系统会提示输入并确认密码，输入密码时，终端不会显示任何字符，以确保安全。

删除名为 newuser 的用户：

```
#sudo userdel  newuser
```

删除名为 newuser 的用户并同时删除其主目录：

```
#sudo userdel  -r newuser
```

1.5. Kdump 机制

Kdump 是基于 kexec 的内核崩溃转储机制，在系统崩溃、死锁或死机时用来转储内存运行参数的一个工具和服务，用来捕获内核崩溃的时候产生的 crash dump。Kdump 是迄今为止最可靠的内核转存机制，最大的优点在于崩溃转储数据可从一个新启动内核的上下文中获取，而不是从已经崩溃内核的上下文。

Kdump 需要两个不同目的的内核，生产内核和捕获内核。生产内核是捕获内核服务的对像：如果系统一旦崩溃，那么正常的内核就没有办法工作了，在这个时候将由 Kdump 产生一个用于捕获当前运行信息的内核，该内核会与相应的 ramdisk（虚拟内存盘：将内存模拟成硬盘的技术）一起组建一个微环境，将此时的内存中的所有运行状态和数据信息收集到一个 dump core 文件中，一旦内存信息收集完成，系统将会自动重启。

Kdump 机制主要包括两个组件：kdump 和 kexec。

kdump 使用 kexec 启动到捕获内核，以很小内存启动以捕获转储镜像。生产内核保留了内存的一部分给捕获内核启动用。由于 kdump 利用 kexec 启动捕获内核，绕过了 BIOS，所以第一个内核的内存得以保留。这是内核崩溃转储的本质。

kexec 是一个快速启动 kernel 的机制，它运行在某一正在运行的 kernel 中，启动一个新的 kernel 而且不用重新经过 BIOS 就可以完成启动。因为一般 BIOS 都会花费很长的时间，尤其是在大型并且同时连接许多外部设备的 Server 上的环境下，BIOS 会花费更多的时间。

kexec 包括 2 个组成部分：一是内核空间的系统调用 kexec_load，负责在生产内核启动时将捕获内核加载到指定地址。二是用户空间的工具 kexec-tools，他将捕获内核的地址传递给生产内核，从而在系统崩溃的时候能够找到捕获内核的地址并运行。

1.5.1. Kdump 命令行配置

1.5.1.1. 安装 Kdump 需要的软件包

软件包名称	软件包说明
kexec-tools	kexec 软件包，kdump 用到的各种工具都在此包中
kernel-debuginfo	用来分析 vmcore 文件

使用 Kdump 服务，需先安装这些工具包。安装命令如下：

```
dnf install kexec-tools kernel-debuginfo
```

1.5.1.2. 配置 grub

Kdump 的使用需要配置 kdump kernel 的内存区域。Kdump 要求操作系统正常使用的时候，不能使用 kdump kernel 所占用的内存，配置这个需要修改 /boot/efi/EFI/kylin/grub.cfg 文件（x86 legacy 固件则需修改/boot/grub2/grub.cfg 文件），修改用到的引导部分，加入 crashkernel。Crashkernel 的格式如下：

```
crashkernel=nn[KMG],high
```

表示在物理内存预留 nn 大小的内存给 kdump 使用。修改完成并重启后，可以通过 cat /proc/cmdline 查看 kernel 启动配置选项，其中已经加入了 crashkernel 项。

1.5.1.3. 启动和查看 Kdump 服务

查看 Kdump 服务命令如下：

```
#systemctl status kdump.service
```

如果 Kdump 服务未开启，使用如下命令来启动 kdump 服务:

```
#systemctl start kdump.service
```

1.6. 获取特权

系统普通用户的权限有不同的限制，某些情况下普通需用需要执行管理员用户权限才能执行的命令，此时可以通过 su 或者 sudo 命令获得管理员权限特权。

1.6.1. su 命令工具

用户使用 su 命令时，需要输入 root 用户密码，验证通过后可以获取 root 的脚本环境。一旦通过 su 命令登入，这个用户的所有操作均视为 root 用户操作。由于 su 可以获取 root 全部权限，并因此获取其他用户的权限，可能存在一定安全问题。因此可以通过管理员组群 wheel 来进行限制。以 root 用户执行以下命令：

```
#usermod -G wheel username
```

当将用户加入 wheel 组群后,可以限制只有这个组群的用户可以使用 su 命令访问。配置 su 的 PAM 可以编辑/etc/pam.d/su 文件，通过添加删除#字符来确认添加或删除相应内容。

```
#auth required pam_wheel.so use_uid
```

上述内容表示管理员组群 wheel 内的用户可以通过 su 访问其他用户。

1.6.2. sudo 命令工具

sudo 命令允许系统管理员让普通用户执行一些或者全部的 root 命令。当可信用户执行 sudo 命令时，需要提供他们自己的用户密码，然后以 root 权限执行命令。

基本的 sudo 命令如下：

```
#sudo command
```

sudo 命令有很大的弹性，只有在/etc/sudoers 文件中被允许的用户可以执行在他们自己的 shell 环境中执行 sudo 命令，而不是 root 的 shell 环境。

配置 sudo 必须通过编辑/etc/sudoers 文件，而且只有管理员用户才可以修改它，必须使用 visudo 编辑。之所以使用 visudo 有两个原因，一是它能够防止两个用户同时修改它；二是它也能进行一些的语法检查。以 root 身份用 visudo 打开配置文件，输入以下内容：

```
    #username ALL=(ALL)    ALL
```

这条信息意思是 username 用户可以以任何主机连接并通过 sudo 执行任何命令。

下面这条信息说明 users 用户可以本地主机可以执行/sbin/shutdown -h now 命令：

```
%users localhost=/sbin/shutdown -h now
```

1.7. 内存分级扩展

1.7.1. 内存分级扩展（etmem）介绍

etmem 内存分级扩展机制，通过 DRAM+内存压缩/高性能存储新介质形成多级内存存储，用于在当前系统下扩展某些应用进程的内存，或将应用进程的内存根据访问频率放置到不同速率的内存介质中，使用 etmem 内存扩展机制，需要以下 3 方面支持：

- 内核支持：etmem_scan.ko 和 etmem_swap.ko
- 客户端程序 etmem
- 服务端程序 etmemd

客户端程序 etmem 配置哪些应用程序需要用哪种内存扩展方法，通过命令行调用将请求发送给服务端程序 etmemd，服务端程序 etmemd 通过和内核交互对应用进程

实现相应的内存扩展策略。

etmem 适用于对内存使用较多，且访问相对不频繁的业务软件，扩展效果较好，比如 MySQL、Redis、Nginx 等，内存扩展操作均在节点内部，不涉及跨节点远端操作。在用户态存储框架的场景中，可通过策略框架的用户态 userswap 功能，用户根据需要提供换入换出接口，使用用户态存储作为交换设备。

1.7.2. 使用方法

两种主要的内存扩展方法适用于不同的内存介质，slide 适用于 nvme 磁盘类，cslide 适用于 AEP 类介质。本使用说明限于硬件原因，只以 slide 举例测试。

1.7.2.1. 使用约束

- etmem 的客户端和服务端需要在同一个服务器上部署，不支持跨服务器通信的场景。
- etmem 仅支持扫描进程名小于或等于 15 个字符长度的目标进程。
- 在使用 AEP 介质进行内存扩展的时候，依赖于系统可以正确识别 AEP 设备并将 AEP 设备初始化为 numa node。并且配置文件中的 vm_flags 字段只能配置为 ht。
- 引擎私有命令仅针对对应引擎和引擎下的任务有效，比如 cslide 所支持的 showhostpages 和 showtaskpages。
- 禁止并发扫描同一个进程。未加载 etmem_scan 和 etmem_swap ko 时，禁止使用/proc/xxx/idle_pages 和/proc/xxx/swap_pages 文件。
- 不能对加锁内存进行换出。
- 若应用程序访问内存频率较高，不建议使用，如 stress 压测工具。

1.7.2.2. 工具参数说明

etmemd 参数说明如下表。

参数	说明
-l 或--log-level	etmemd 日志级别，取值范围 0-3
-s 或--socket	etmemd 监听的名称，用于与客户端交互，107 个字符之内的字符串
-h 或--help	帮助信息
-m 或--mode-systemctl	etmemd 作为 service 被拉起时，命令中可以使用此参数来支持 fork 模式启动

etmem 参数说明如下表。

参数	说明
-f 或--file	指定对象的配置文件，add、del 子命令必须包含该参数
-s 或--socket	与 etmemd 服务端通信的 socket 名称，需要与 etmemd 启动时指定的保持一致，add、del 子命令必须包含
OBJECT	参数可选值为 project、obj、engine，请参考帮助信息
COMMAND	参数可选值为 add、del、start、stop、show、eng_cmd、help，请参考帮助信息

除上述两个工具，还有一个重要的配置文件，该配置文件需要管理员预先规划哪些进程需要做内存扩展，将进程信息配置到 etmem 配置文件中，并配置内存扫描的周期、扫描次数、内存冷热阈值等信息。配置文件的示例文件在源码包中，放置在/etc/etmem 文件路径下，按照功能划分为 3 个示例文件：/etc/etmem/cslide_conf.yaml、/etc/etmem/slide_conf.yaml、/etc/etmem/thirdparty_conf.yaml。配置文件中的字段说明如下表。

配置项	配置项含义	是否必需	是否有参数	参数范围	示例说明
[project]	project 公用配置段起始标识	否	否	NA	project 参数的开头标识,表示下面的参数直到另外的[xxx]或文件结尾为止的范围内均为 project section 的参数
name	project 的名字	是	是	64 个字以内的字符串	用来标识 project，engine 和 task 在配置时需要指定要挂载

					到的 project
loop	内存扫描的循环次数	是	是	1~120	loop=3 //扫描 3 次
interval	每次内存扫描的时间间隔	是	是	1~1200	interval=5 //每次扫描之间间隔 5s
sleep	每个内存扫描+操作的大周期之间时间间隔	是	是	1~1200	sleep=10 //每次大周期之间间隔 10s
sysmem_threshold	slide engine 的配置项，系统内存换出阈值	否	是	0~100	sysmem_threshold=50 // 系统内存剩余量小于 50%时，etmem 才会触发内存换出

swapcache_high_wmark	slide engine 的配置项，swacache 可以占用系统内存的比例，高水线	否	是	1~100	swapcache_high_wmark=5 //swapcache 内存占用量可以为系统内存的 5%，超过该比例，etmem 会触发 swapcache 回收
					注：swapcache_high_wmark 需要大于 swapcache_low_wmark

swapcache_low_wmark	slide engine 的配置项，swacache 可以占用系统内存的比例，低水线	否	是	[1~swapcache_high_wmark)	swapcache_low_wmark=3 // 触发 swapcache 回收后，系统会将 swapcache 内存占用量回收到低于 3%
[engine]	engine 公	否	否	NA	engine 参数的开头标识，表示下

	用配置段起始标识				面的参数直到另外的[xxx]或文件结尾为止的范围内均为 engine section 的参数
project	声明所在的 project	是	是	64 个字以内的字符串	已经存在名字为 test 的 project，则可以写为 project=test
engine	声明所在的 engine	是	是	slide/cslide/thridparty	声明使用的是 slide 或 cslide 或 thirdparty 策略
node_pair	cslide engine 的配置项，声明系统中 AEP 和 DRAM 的 node pair	engine 为 cslide 时必须配置	是	成对配置 AEP 和 DRAM 的 node 号，AEP 和 DRAM 之间用逗号隔开，每对 pair 之间用分号隔开	node_pair=2,0;3,1
hot_threshold	cslide engine 的配置项，声明内存冷热水线的阈值	engine 为 cslide 时必须配置	是	大于等于 0，小于等于 INT_MAX 的整数	hot_threshold=3 //访问次数小于 3 的内存会被识别为冷内存
node_mig_quota	cslide engine 的配置项，流控，声明每次 DRAM	engine 为 cslide 时必	是	大于等于 0，小于等于 INT_MAX 的整数	node_mig_quota=1024 //单位为 MB，AEP 到 DRAM 或 DRAM 到 AEP 搬迁一次最大 1024M

	和 AEP 互相迁移时单向最大流量	须配置			
node_hot_reserve	cslide engine 的配置项，声明 DRAM 中热内存的预留空间大小	engine 为 cslide 时必须配置	是	大于等于 0，小于等于 INT_MAX 的整数	node_hot_reserve=1024 //单位为 MB，当所有虚拟机热内存大于此配置值时，热内存也会迁移到 AEP 中
eng_name	thirdparty engine 的配置项，声明 engine 自己的名字，供 task 挂载	engine 为 thirdparty 时必须配置	是	64 个字以内的字符串	eng_name=my_engine // 对此第三方策略 engine 挂载 task 时，task 中写明 engine=my_engine
libname	thirdparty engine 的配置项，声明第三方策略的动态库的地址，绝对地址	engine 为 thirdparty 时必须配置	是	256 个字以内的字符串	libname=/user/lib/etmem_fetch/code_test/my_engine.so
ops_name	thirdparty engine 的配置项，声明第三方策	engine 为 thirdpa	是	256 个字以内的字符串	ops_name=my_engine_ops //第三方策略实现接口的结构体的名字

	略的动态库中操作符号的名字	rty 时必须配置			
engine_private_key	thirdparty engine 的配置项，预留给第三方策略自己解析私有参数的配置项，选配	否	否	根据第三方策略私有参数自行限制	根据第三方策略私有 engine 参数自行配置
[task]	task 公用配置段起始标识	否	否	NA	task 参数的开头标识，表示下面的参数直到另外的[xxx]或文件结尾为止的范围内均为 task section 的参数
project	声明所挂的 project	是	是	64 个字以内的字符串	已经存在名字为 test 的 project，则可以写为 project=test
engine	声明所挂的 engine	是	是	64 个字以内的字符串	所要挂载的 engine 的名字
name	task 的名字	是	是	64 个字以内的字符串	name=background1 // 声明 task 的名字是 backgound1
type	目标进程识别的方式	是	是	pid/name	pid 代表通过进程号识别，name 代表通过进程名称识别
value	目标进程识别的具体字段	是	是	实际的进程号/进程名称	与 type 字段配合使用，指定目标进程的进程号或进程名称，由使用者保证配置的正确及唯一性
T	engine 为 slide 的 task 配置	engine 为	是	0~loop * 3	T=3 //访问次数小于 3 的内存会被识别为冷内存

	项，声明内存冷热水线的阈值	slide 时必须配置			
max_threads	engine 为 slide 的 task 配置项，etmemd 内部线程池最大线程数，每个线程处理一个进程/子进程的内存扫描+操作任务	否	是	1~2 * core 数 + 1，默认为 1	对外部无表象，控制 etmemd 服务端内部处理线程个数，当目标进程有多个子进程时，配置越大，并发执行的个数也多，但占用资源也越多
vm_flags	engine 为 cslide 的 task 配置项，通过指定 flag 扫描的 vma，不配置此项时扫描则不会区分	否	是	256 长度以内的字符串，不同 flag 以空格隔开	vm_flags=ht //扫描 flags 为 ht（大页）的 vma 内存
anon_only	engine 为 cslide 的 task 配置项，标识是否只扫描匿	否	是	yes/no	anon_only=no //配置为 yes 时只扫描匿名页，配置为 no 时非匿名页也会扫描

	名页				
ign_host	engine 为 cslide 的 task 配置项，标识是否忽略 host 上的页表扫描信息	否	是	yes/no	gn_host=no //yes 为忽略，no 为不忽略
task_private_key	engine 为 thirdparty 的 task 配置项，预留给第三方策略的 task 解析私有参数的配置项，选配	否	否	根据第三方策略私有参数自行限制	根据第三方策略私有 task 参数自行配置

swap_threshold	slide engine 的配置项，进程内存换出阈值	否	是	进程可用内存绝对值	swap_threshold=10g //进程占用内存在低于 10g 时不会触发换出。
					当前版本下，仅支持 g/G 作为内存绝对值单位。与 sysmem_threshold 配合使用，仅系统内存低于阈值时，进行白名单中进程阈值判断

swap_flag	slide engine 的配置项，进程指定内存换出	否	是	yes/no	swap_flag=yes//使能进程指定内存换出

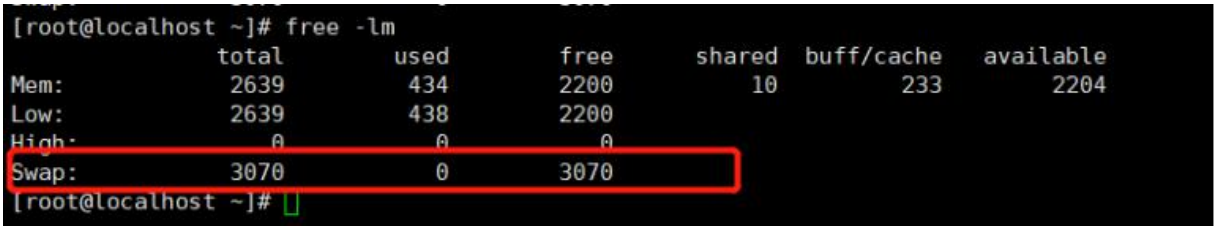
1.7.2.3. 测试方法

(1) 创建 swap 分区

假设用于 swap 分区的硬盘为 sdc，执行以下命令创建 swap 分区。

命令	作用
# fdisk /dev/sdc	新建 sdc1 分区用于 swap 分区
# free -lm	查看当前 swap 分区情况
# swapoff -a	关闭当前 swap 分区
# mkswap /dev/sdc1	格式化 swap 分区
# swapon /dev/sdc1	启用新的 swap 分区
# free -lm	查看新 swap 分区

执行上面表格中的最后一个 free -lm 结果如下。



可看出当前 swap 分区未被使用。

(2) 创建测试应用程序

创建 stress 应用程序，脚本中分配 1.9G 内存（该值需要根据系统中的内存大小和配置文件中设定的 sysmem_threshold 值进行调整，该值设置太小可能导致 swap 分区不被使用），内容如下，请根据实际使用情况指定应用程序，stress.c 文件内容如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define SIZE 1024 * 1024 * 1946  //请根据实际情况修改此值
int main() {
```

```
    char *memory;
    // 分配 1.9 GiB 的内存
    memory = (char *)malloc(SIZE);
    if (memory == NULL) {
        fprintf(stderr, "Error: Failed to allocate memory\n");
        return 1;
    }
    // 访问分配的内存
    memset(memory, 'A', SIZE);
    sleep(300);
    // 释放内存
    free(memory);
    printf("Memory allocation and access test completed successfully.\n");
    return 0;
}
```

编译并执行该脚本：

```
# gcc -o stress stress.c
# ./stress &
# pgrep -l stress
```

(3) 安装 etmem

```
先确认是否已安装 etmem 包
#rpm -qa |grep etmem
若上述命令无 etmem 结果，请安装 etmem 包，否则跳过安装
#yum install etmem
```

(4) 修改 etmem 配置文件

在运行 etmem 进程之前，需要管理员预先规划哪些进程需要做内存扩展，将进程信息配置到 etmem 配置文件中，并配置内存扫描的周期、扫描次数、内存冷热阈值等信息。

vi /etc/etmem/slide_conf.yaml

```
[project]
name=test
scan_type=page
loop=1
```

```
interval=1
sleep=1
sysmem_threshold=50
swapcache_high_wmark=50
swapcache_low_wmark=20

[engine]
name=slide
project=test

[task]
project=test
engine=slide
name=background_slide
type=name
value=stress
T=2
```

(5) 加载内核模块

```
modprobe etmem_scan
modprobe etmem_swap
```

(6) 启动 etmemd

```
etmemd -l 0 -s etmemd_socket
```

启动成功没有报错，另外还支持编写 service 配置文件，来启动 etmemd，必须使用-m 参数来指定此模式，命令为：

```
etmemd -l 0 -s etmemd_socket -m
```

(7)创建工程

```
etmem obj add -f /etc/etmem/slide_conf.yaml -s etmemd_socket
```

创建成功没有报错。

(8)查询工程

```
etmem project show -n test -s etmemd_socket
```

命令结果如下图。

```
[root@localhost ~]# etmem project show -n test -s etmemd_socket
project: test
number   type     value            name             engine           started
1        name     stress           background_slide slide            false
```

(9)启动工程

```
etmem project start -n test -s etmemd_socket
```

启动成功没有报错。

(10)查看工程，命令同步骤(7)，结果如下

```
[root@localhost ~]# etmem project show -n test -s etmemd_socket
project: test
number   type     value            name             engine           started
1        name     stress           background_slide slide            true
[root@localhost ~]#
```

(11)运行一段时间后查看 swap 内存使用情况

```
# free -m
```

可以看到 swap 分区被使用，如下图：

```
[root@localhost ~]# free -m
              total        used        free      shared  buff/cache   available
Mem:           2639         523        2106          10         241        2115
Swap:          3070        1933        1137
[root@localhost ~]# etmem project show -n test -s etmemd_socket
```

(12)删除工程

不想使用 etmem 时，使用下面命令删除工程，删除工程的会自动停止工程再删除。

```
etmem obj del -f /etc/etmem/slide_conf.yaml -s etmemd_socket
```

2. 基本开发环境

2.1. Qt-5.14.2

Qt 是一个跨平台的桌面、嵌入式和移动应用程序开发框架。只需重新编译即可将现有的桌面或嵌入式应用程序带到移动设备中。有很强的图形功能和性能。Qt5 是 Qt 的最新版本，开发人员能够以直观的用户界面针对多个目标开发应用程序。通过 Qt5 中改进的 JavaScript 和 QML 支持，开发可以更加高效和灵活，同时仍支持 C++和 Qt

Widget。Qt5 与 Qt4 高度兼容，并借助模块化的代码库和 Qt Platform Abstraction，增强了代码的可移植性。

当前更新至版本 5.14.2，更多新特性请查阅 https://doc.qt.io/archives/qt-5.14

2.2. GCC

GCC 是由 GNU 开发的编程语言编译器，支持多种语言的编译，例如 C，C++，Objective-C，Java 等，同时包含这些语言的库文件。GCC 是一种开源的开发工具，支持多种体系架构，易于扩展和测试。

主要特性包括：

- 支持 GNU 标准；
- 编译器基于 GPL 标准；
- 具有不断更新的运行时库，调试效率高；

当前更新至版本 7.3（x86 和 ARM 架构）、8.3（Loongarch 架构）详细用法或其他资料请查阅 https://gcc.gnu.org/。

2.3. GDB

GDB 是 GNU 代码调试器，允许查看程序内部执行流程，或者程序在发生异常时的状态。GDB 的功能主要包括：

执行一些能够影响程序运行结果的操作；

在指定的条件下停止程序；

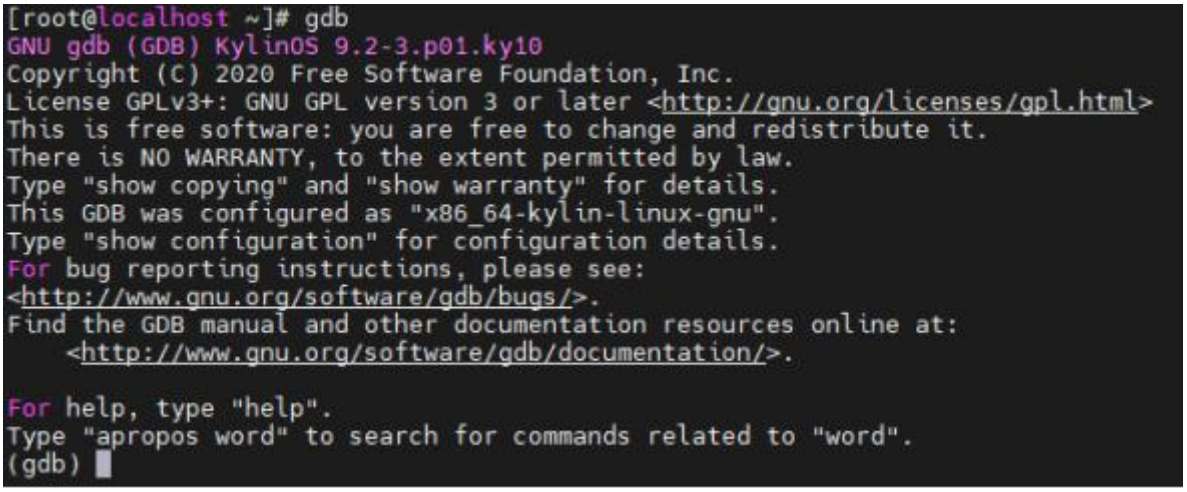
在程序停止运行时，检查此时程序内部发生了什么；

修改程序，以验证程序 bug 对程序的影响，同时了解到程序中许多其他的内容，例如变量取值等。

GDB 支持以下编程语言：

Ada、Assembly、C、C++、D、Fortran、Go、Objective-C、OpenCL、Modula-2、Pascal、Rust

终端控制台输入命令”gdb”，即可打开 GDB 调试工具。



当前更新至版本 9.2（x86 和 ARM 架构）、8.3（Loongarch 架构），详细用法或其他资料请查阅 http://gnu.org/software/gdb/。

2.4. Python3

Python 是一种清晰而强大的面向对象编程语言，可与 Perl、Ruby、Scheme 或 Java 相媲美。

Python 的一些显著特性:语法简洁，程序易于阅读；程序运行简单，这使得 Python 成为许多编程任务的理想选择，同时又不影响可维护性；附带一个大型标准库，支持许多常见的编程任务，如连接网络服务器、用正则表达式搜索文本、读取和修改文件等；Python 的交互模式使得测试简短的代码片段变得很容易，python 开发环境叫做 IDLE。

通过添加以编译语言(如 C 或 C++)实现的新模块，可以轻松地进行扩展。

也可以嵌入到应用程序中以提供可编程接口。

支持多个系统平台，包括 Mac OS X，Windows，Linux 和 Unix，非官方版本也可用于 Android 和 iOS。

Python 是一款免费软件。下载或使用 Python，或者将它包含在应用程序中不需要任何费用，Python 也可以被自由修改和重新发布。

Python 的一些编程语言特性包括:

有多种基本数据类型可用:number(floating point,complex 和 unlimited-length

long integers)、strings(包括 ASCII 和 Unicode)、lists 和 dictionaries。

Python 支持带有类和多重继承的面向对象编程。

代码可以分成模块和包。

该语言支持引发和捕获异常，从而实现更清晰的错误处理。

数据类型是强类型和动态类型。混合不兼容的类型(例如，试图添加一个字符串和一个数字)会导致引发异常，从而更快地发现错误。

Python 包含高级编程特性，如 generators 和 list comprehensions。

Python 支持自动内存管理，开发人员不必手动分配和释放代码中的内存。

当前更新至版本 3.7.9，详细资料请查阅 https://www.python.org/。

2.5. Openjdk

Openjdk 作为 GPL 许可的 Java 平台开源化实现，由 Sun 公司开发，提供了一个 java 的运行环境，支持 Solaris、Linux、Mac OS X 或 Windows 多种操作系统。

新版本的新特性主要有：Lambda 表达式和 Stream API；时间与日期 API；构造器引用；红黑树的使用使运行速度更快；减少空指针异常等。

当前更新至版本 1.8.0 和 11.0，详细资料请查阅 http://openjdk.java.net。

3. 安装和管理软件

dnf 是新一代的软件包管理器，其在设计上克服了前代包管理器的瓶颈，显著提升了用户体验、减少了内存占用、改进了依赖分析的效率，并加快了运行速度。dnf 利用 rpm、libsolv 和 hawkey 等库来执行其包管理操作，与 yum 相比，dnf 提供了更先进的依赖解决算法和资源管理策略，同时保持了与 yum 的高度兼容性，使得两者可以在同一系统中共存和使用。

这种进化不仅提高了包管理的性能，还增强了系统的稳定性，使得用户和系统管理员在处理软件包的安装、更新和删除时，能够享受到更加流畅和可靠的操作体验。

3.1. 检查和升级软件包

3.1.1. 软件包升级检查

查看系统里已经安装的软件包有哪些可以升级可以执行以下命令，以 X86 平台示例如下：

```
[root@localhost ssh]# dnf check-update
上次元数据过期检查：2:09:08 前，执行于 2022年07月20日 星期三 09时54分49秒。

anaconda.x86_64                          33.19-31.p12.ky10                      ks10-a
anaconda-core.x86_64                     33.19-31.p12.ky10                      ks10-a
anaconda-tui.x86_64                      33.19-31.p12.ky10                      ks10-a
```

示例说明：

PackageKit——软件包名称；

x86_64——该软件包支持的 CPU 架构；

33.19-31.p12.ky10——可升级的软件包版本；

ks10-adv-os——可升级的软件包所存储仓库。

3.1.2. 升级软件包

dnf 支持一次升级单个/批量软件包，并同时安装/更新相应的依赖包。

1.升级单一软件包命令：

```
#dnf update {package_name}
```

升级 kernel 软件包命令为例：

```
[root@localhost ssh]# dnf update kernel
上次元数据过期检查：2:01:20 前，执行于 2022年07月20日 星期三 09时54分49秒。
依赖关系解决。
================================================================================
 Package                 Architecture      Version                  Repository          Size
================================================================================
安装：
 kernel                  x86_64            4.19.90-51.0.v2207.ky10  ks10-adv-os-koji   749 k
安装依赖关系：
 kernel-core             x86_64            4.19.90-51.0.v2207.ky10  ks10-adv-os-koji    38 M
 kernel-modules          x86_64            4.19.90-51.0.v2207.ky10  ks10-adv-os-koji    21 M
 kernel-modules-extra    x86_64            4.19.90-51.0.v2207.ky10  ks10-adv-os-koji   2.2 M

事务概要
================================================================================
安装  4 软件包

总下载：62 M
安装大小：86 M
确定吗？[y/N]：
```

上述输出的说明如下：

a)Package：用户需要下载升级的软件包和依赖软件包。

b)Architecture：该软件包所属的架构。

c)Version：软件包升级后的版本。

d)Repository：可升级软件包所属仓库。

e)Size：软件包大小。

f)dnf 默认会显示升级软件包的基本信息，并提示是否确认安装，用户可以在使用 dnf 命令时添加参数–y，效果等同于出现 Is this ok [y/N]:时输入 y。

g)安装过程中如果出现错误导致安装过程终止，可以使用 dnf history 命令查看详细描述。

如果需要安装一组软件包，可以以 root 用户执行命令：

```
#dnf groupupdate group_name
```

2.批量升级软件包及其依赖

如果需要升级系统所有软件包，可以使用以下命令：

```
#dnf update
```

3.1.3. 升级系统

利用系统光盘与 dnf 离线升级系统,dnf 可与 yum 使用相同的配置文件,即配置 dnf 源可直接对/etc/yum.repos.d/中的.repo 文件进行编辑。当系统处于离线状态或者无法访问官方更新源时，可以利用更新的系统光盘创建本地源并进行升级。步骤如下：

1.创建系统光盘挂载目录，以 root 用户执行：

```
#mkdir {mount_dir}
```

2.将系统安装光盘挂载至该目录，以 root 用户执行

```
#mount -o loop {iso_name} {mount_dir}
```

3.编辑/etc/yum.repos.d/new.repo 配置文件以配置本地源：

```
[local repo]     #仓库标识
name=local-repo     #仓库名称
```

```
baseurl=file:/// mount_dir   #仓库路径
enabled=1      #启用此仓库
gpgcheck=0      #禁用 gpg 检验功能
```

4.更新 dnf 源并进行升级，以 root 用户执行：

```
#dnf update
```

5.升级成功后，卸载系统光盘挂载目录：

```
#umount mount_dir 或者 rmdir mount_dir
```

如果不再使用这个 dnf 源进行安装和升级，可以以 root 用户删除文件：

```
#rm /etc/yum.repos.d/new.repo
```

3.2. 管理软件包

dnf 提供了完整操作系统软件包管理功能，包括检索、查看信息、安装和删除。

3.2.1. 检索软件包

执行 dnf search 命令可以检索软件包，例如检索包含“mesh”字段的软件包，以 X86 平台示例如下：

```
#dnf search mesh
```

如果 dnf 检测的结果繁多，可以通过 shell 本身的 grep 或者正则表达式进行过滤。

3.2.2. 安装包列表

显示已安装和可安装的软件包列表可以执行以下命令：

```
#dnf list all
```

显示包括某些字符的已安装和可安装软件包列表可以执行以下命令：

```
#dnf list glob_expression...
```

显示 abrt 相关软件包列表的命令如下：

```
#dnf list abrt-addon\* abrt-plugin\*
```

显示包括某些字符的已安装软件包列表可以执行以下命令：

```
#dnf list installed glob_expression...
```

显示包括 krb 的所有已安装软件包示例如下：

```
#dnf list installed "krb?-*"
```

显示包括某些字符的可安装软件包列表可以执行以下命令：

```
#dnf list available glob_expression...
```

显示所有可用的 gstreamer plug-ins 软件包列表：

```
#dnf list available gstreamer\*plugin\*
```

查看软件仓库

成功注册后，可使用 dnf 来管理软件包。

查看可用的软件仓库可以使用以下命令：

```
#dnf repolist
```

如果想显示更多信息可以加上-v 选项，或者用 dnf repoinfo 命令输出信息。

```
#dnf repolist -v
```

```
#dnf repoinfo
```

如果需要显示所有可用和不可用的软件仓库，可以使用以下命令：

```
#dnf repolist all
```

3.2.3. 显示软件包信息

显示一个或多个软件包可以使用以下命令：

```
#dnf info package_name...
```

显示软件包 abrt 详细信息的命令：

```
#dnf info abrt
```

显示软件包 yum 详细信息的命令：

```
#dnf info yum
```

3.2.4. 安装软件包

用户可以以 root 用户使用以下命令安装软件包

```
#dnf install package_name
```

安装 sqlite 的 x86_64 架构的软件包示例：

```
#dnf install sqlite.x86_64
```

除了安装软件包，还可以安装具体的二进制文件，您可以输入文件地址，以 root 用户执行安装：

```
#dnf install /usr/sbin/named
```

安装命令如下：

```
#dnf install httpd
```

安装本地软件包 abrt 示例如下：

```
#dnf localinstall path
```

3.2.5. 下载软件包

在执行安装流程中，显示以下选项是：

```
. . .
Total size: 1. 2 M
Is this ok [y/N]:
. . .
```

输入 y，可以执行软件包下载。

3.2.6. 删除软件包

删除软件包可以执行以下命令：

```
dnf remove package_name...
```

删除 totem 软件包示例：

```
dnf remove totem
```

3.3. 管理软件包组

软件包组可以搜集一系列特定功能软件包，比如系统工具和视频软件包组。安装软件包组可以一起安装其依赖。

3.3.1. 软件包组列表

Summary 选项可以显示软件包可用组的数量：

```
dnf groups summary
```

以下为输出示例：

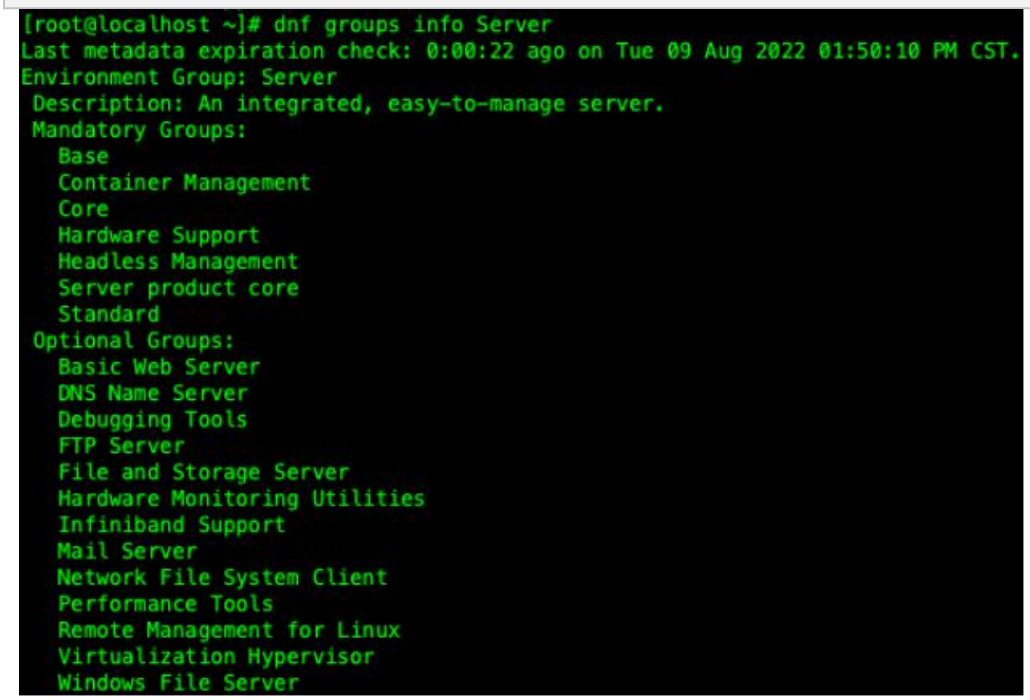
```
#dnf groups summary
可用组： 8
```

显示某个软件包组的全部信息可以用以下命令：

```
#dnf groups info glob_expression...
```

以下为 Server 组输出示例：

```
#dnf groups info Server
```



3.3.2. 安装软件包组

每个软件包组都有自己的组 id，要显示包组 id 可以使用以下命令：

```
#dnf group list ids
```

查找开发软件包组列表的示例：

```
#dnf groups list ids deve\*
```

```
[root@localhost ~]# dnf group list ids deve\*
Last metadata expiration check: 0:12:05 ago on Tue 09 Aug 2022 01:50:10 PM CST.
Available Groups:
   Development Tools (development)
```

软件包组的安装可以通过软件包组名称安装，也可通过包组 id 安装。

```
#dnf group install "group name"
#dnf group install groupid
```

也可用通过以下两种命令安装：

```
#dnf install @group
#dnf install @^group
```

下面是 4 种安装开发工具软件分组的示例：

```
#dnf group install "Development Tools"
#dnf group install development
#dnf install @"Development Tools"
#dnf install @development
```

3.3.3. 删除软件包组

可以通过软件包组名或者软件包组 id 删除软件包。以 root 权限执行：

```
#dnf group remove group_name
#dnf group remove groupid
```

如果软件分组有@标签，也可用以下命令删除。以 root 身份执行：

```
#dnf remove @group
#dnf remove @^group
```

删除 KDE 桌面软件分组示例：

```
#dnf group remove "Development Tools"
#dnf group remove development
#dnf remove @"Development Tools"
#dnf remove @development
```

3.4. 软件包操作记录管理

dnf 可以使用 dnf history 命令进行管理操作。

3.4.1. 查看操作

显示以往 20 条 dnf 操作记录，可以使用以下命令。以 root 权限执行：

```
#dnf history list 1..20
```

ID	命令行	日期和时间	操作	更改
20	install libcap-devel	2021-06-08 14:02	I, U	2
19	update yum	2021-05-31 14:01	I, U	9 EE
18	install bind-utils	2021-05-28 09:57	Install	10
17	install net-tools	2021-05-25 09:56	Install	1
16	install docker	2021-05-24 15:15	Install	2 EE
15	install zlib-devel	2021-05-24 13:29	Install	1
14	install php-devel	2021-05-24 13:26	Install	8
13	install telnet	2021-05-20 08:54	Install	1
12	update	2021-05-17 13:49	E, I, U	17 EE
11	update	2021-05-13 08:43	Upgrade	13 EE
10	install php-mbstring	2021-04-30 17:33	Install	2
9	install php-mysqli	2021-04-30 17:32	Install	2
8	install php-json	2021-04-30 17:32	Install	1
7	install php-xml	2021-04-30 17:32	Install	1
6	install php-fpm	2021-04-30 17:25	Install	2
5	install -y php	2021-04-30 17:23	Install	3
4	install -y mariadb-server	2021-04-30 16:01	Install	6
3	install -y httpd	2021-04-30 15:55	Install	7
2	update	2021-04-30 15:46	Upgrade	19 EE
1		2021-04-30 15:10	Install	742 EE

如果想显示某一部分 dnf 操作记录，可以使用以下命令。以 root 权限执行：

```
#dnf history list start_id. . end_id
```

显示过去 5 条 dnf 信息示例如下：

```
#dnf history list 1..5
```

ID	命令行	日期和时间	操作	更改
5	install -y php	2021-04-30 17:23	Install	3
4	install -y mariadb-server	2021-04-30 16:01	Install	6
3	install -y httpd	2021-04-30 15:55	Install	7
2	update	2021-04-30 15:46	Upgrade	19 EE
1		2021-04-30 15:10	Install	742 EE

以上 dnf history list 输出显示内容说明如下：

ID——识别特定记录的标示数；

Command line——简要描述操作内容；

Date and time——该条记录的日期和时间；

Action(s)——描述操作类型；

Altered——记录操作影响的条目数。

下表是 Action 的不同说明：

Action	缩写	描述
Downgrade	D	降级软件包
Erase	E	删除软件包
Install	I	安装软件包
Obsoleting	O	软件包标注废弃
Reinstall	R	软件包重装
Update	U	升级软件包

3.4.2. 审查操作

需要显示某条操作记录的具体综述信息，可以执行以下命令：

```
#dnf history {id}
```

其中 id 是操作的 id。

如果需要显示某条操作记录的详细信息，可以使用以下命令：

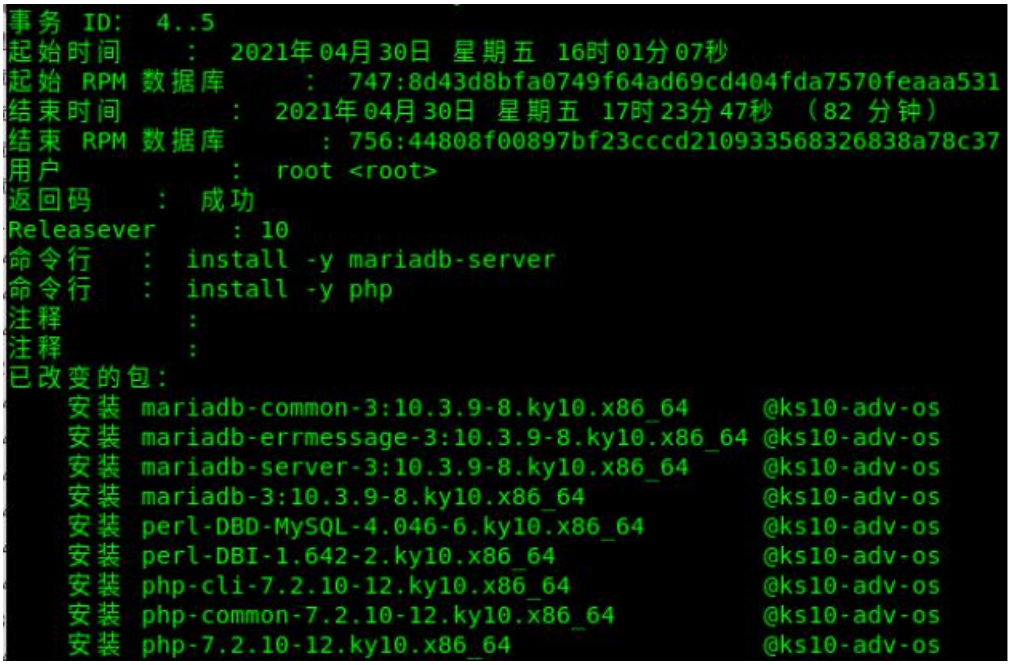
```
#dnf history info {id}
```

如果需要显示某一阶段操作记录的详细信息，可以使用以下命令：

```
#dnf history info start_id. . end_id
```

示例如下：

```
#dnf history info 4 . . 5
```



3.4.3. 恢复与重复操作

如果想要撤销某个 dnf 操作，可以以 root 权限执行以下操作：

```
#dnf history undo {id}
```

如果需要重复某个 dnf 操作，可以以 root 权限执行以下操作：

```
#dnf history redo {id}
```

4. KVM 虚拟化

KVM 是（Kernel-based Virtual Machine）的简称，是适用于支持硬件虚拟化的 arm、x86_64 和 loongarch64 体系结构的全虚拟化解决方案。可以直接使用 QEMU 工具或使用基于 libvirt 工具栈来管理 VM Guest（虚拟机）、虚拟储存和虚拟网络。QEMU 工具包括 qemu-kvm、QEMU 监视器、qemu-img 和 qemu-ndb 等。libvirt 工具栈包括 libvirt 本身，以及 virsh、virt-manager、virt-install 和 virt-viewer 等基于 libvirt 的应用程序。

4.1. 虚拟化基础功能

4.1.1. 轻型虚拟机介绍

轻型虚拟机是一种最小化的虚拟机，相比传统虚拟机，轻型虚拟机移除了对于 PCI 设备和 ACPI 的支持，通过 MMIO 的方式提供虚拟 IO 设备支持，使用内核直接引导，因此能够提供更快的启动速度和更少的内存开销。

目前支持在 X86_64 架构的 HostOS 系统上使用轻型虚拟机。

4.1.2. 轻型虚拟机使用步骤

(1) 准备工作

轻型虚拟机使用内核直接引导，因此需要准备好 linux 内核及装有操作系统根分区内容的 rootfs 虚拟磁盘。

内核在编译配置中需要启用 VIRTIO_MMIO 及 VIRTIO 设备相关配置：

```
CONFIG_BLK_MQ_VIRTIO=y
CONFIG_VIRTIO_VSOCKETS=y
CONFIG_VIRTIO_VSOCKETS_COMMON=y
CONFIG_VIRTIO_BLK=y
CONFIG_VIRTIO_NET=y
CONFIG_VIRTIO_CONSOLE=y
CONFIG_VIRTIO=y
CONFIG_VIRTIO_MENU=y
CONFIG_VIRTIO_BALLOON=y
CONFIG_VIRTIO_MMIO=y
CONFIG_VIRTIO_MMIO_CMDLINE_DEVICES=y
```

rootfs 虚拟磁盘需要使用 qcow2 或 raw 格式，可以使用操作系统发行版提供的 rootfs 虚拟磁盘，或者自行制作，将操作系统根分区的内容拷贝到虚拟磁盘中。

(2) 启动轻型虚拟机

使用 qemu-kvm 命令启动轻型虚拟机，机器类型使用 microvm，参数中需要指明使用的 rootfs 磁盘文件、内核文件以及内核命令行参数，下方为使用 qcow2 格式的 rootfs.qcow2 磁盘文件，内核文件为 bzImage 情况下的命令行内容。

```
# qemu-kvm \
```

```
-M microvm \
-enable-kvm \
-cpu host \
-smp 2 \
-m 1024m \
-nodefaults \
-no-user-config \
-nographic \
-no-reboot \
-device virtio-serial-device \
-chardev stdio,id=virtiocon0 \
-device virtconsole,chardev=virtiocon0 \
-drive id=root,file=rootfs.qcow2,format=qcow2,if=none \
-device virtio-blk-device,drive=root \
-kernel bzImage \
-append "console=hvc0 root=/dev/vda rw acpi=off reboot=t panic=-1"
```

4.2. libvirt 使用

4.2.1. CPU 热插拔

在线增加(热插)虚拟机 CPU 是指在虚拟机处于运行状态下，为虚拟机热插 CPU 而不影响虚拟机正常运行的方案。当虚拟机内部业务压力不断增 大，会出现所有 CPU 均处于高负载的情形。为了不影响虚拟机内的业务正 常运行，可以使用 CPU 热插功能，提升虚拟机的计算能力。需注意的是：当前版本 ARM 架构下不支持 CPU 热拔。

操作步骤：

```
#virsh setvcpus  <domain> <count> [--config] [--live]
```

参数如下表所示。

参数	描述
domain	指定虚拟机名称
count	目标 CPU 数目，即热插后的虚拟机 CPU 数目
--config	虚拟机下次启动时仍有效

--live	在线生效

以下步骤在虚拟机关机状态下执行：

```
#virsh setvcpus --domain --maximum num --conf
```

以下步骤可在虚拟机开机状态下执行：

CPU 热插命令:

```
#virsh setvcpus --domain num --live
```

CPU 热拔命令：

```
#virsh setvcpus --domain num --live
```

4.2.2. CPU 型号配置

虚拟机支持的 cpu 模式有三种，分别为 host-passthrough，host-model，custom 模式。coustom 模式下虚拟机 CPU 指令集数最少，故性能相对最差，但是它在热迁移时跨不同型号 CPU 的能力最强。此外，custom 模式下支持用户添加额外的指令集。当前系统在 arm 架构下新增了对 FT-2000+、Kunpeng-920、Tengyun-S2500cpu 型号的支持。

在 libvirt 中的使用方式如下：

```
<cpu mode='custom' match='exact' check='none'/>
  <model fallback='allow'>Tengyun-S2500</model>
</cpu>
```

model 中字符代表 cpu 型号。

4.2.3. 内存热插拔

使用方法：

内存热扩容使用命令：

```
#virsh setmem  <domain> <size> [--config] [--live]
```

内存热缩容使用命令：

```
#virsh setmem  <domain> <size> [--config] [--live]
```

参数见下表。

参数	描述
domain	指定虚拟机名称
size	目标内存大小，即热插后的虚拟机内存大小
--config	虚拟机下次启动时仍有效
--live	在线生效

注：虚拟机开机前，需设定热插内存上限，改变在虚拟机中的最大内存分配限制。

```
#virsh setmaxmem  <domain> <size>
```

4.2.4. 网卡热插拔

使用方法：

需提前准备 interface.xml 文件

```
<interface type='network'>
  <mac address='52:54:00:fd:2e:86'/>
  <source network='default'/>
  <model type='virtio'/>
</interface>
```

网卡热插命令：

```
#virsh attach-device <domain> <file> [--persistent] [--config] [--live]
[--current]
```

网卡热拔命令：

```
#virsh detach-device <domain> <file> [--persistent] [--config] [--live]
[--current]
```

热插拔参数说明见下表。

参数	描述
domain	指定虚拟机名称
file	指定目标 xml 文件
--config	虚拟机下次启动时仍有效
--live	在线生效
--persistent	让实时更改持久化
--current	影响当前虚拟机

4.2.5. 磁盘热插拔

使用方法：

创建任意大小磁盘

```
#qemu-img create -f qcow2 test.qcow2 10G
Formatting 'test.qcow2', fmt=qcow2 size=10737418240 cluster_size=65536 lazy_refcounts=off refcount_bits=16
```

准备 disk.xml 文件

```
<disk type='file' device='disk'>
  <driver name='qemu' type='qcow2'/>
  <source file='/var/lib/libvirt/images/test.qcow2'/>  /*该位置为磁盘文件绝对路径*/
  <target dev='hda' bus='ide'/>
</disk>
```

磁盘热插命令：

```
#virsh attach-device <domain> <file> [--persistent] [--config] [--live] [--current]
```

磁盘热拔命令：

```
#virsh detach-device <domain> <file> [--persistent] [--config] [--live] [--current]
```

参数见下表。

参数	描述
domain	指定虚拟机名称
file	指定目标 xml 文件
--config	虚拟机下次启动时仍有效
--live	在线生效
--persistent	让实时更改持久化
--current	影响当前虚拟机

4.2.6. Guest-Idle-Haltpoll

Guest-Idle-Haltpoll 技术是指当虚拟机 vCPU 空闲时，不立刻执行 WFx/HLT 并发生 VM-exit，而是在虚拟机内部轮询（polling）一段时间。在该时间段内，其他共享

LLC 的 vCPU 在该 vCPU 上的任务被唤醒不需要发送 IPI 中断，减少了发送和接收处理 IPI 的开销及虚拟机陷出（VM-exit）的开销，从而降低任务唤醒的时延。

Guest-Idle-Haltpoll 技术适用于频繁睡眠和唤醒的业务场景，通过优化 IPI 中断性能，可以提升业务性能。

注： 由于 vCPU 在虚拟机内部执行 idle-haltpoll 会增加 vCPU 在宿主机的 CPU 开销，所以开启该特性建议 vCPU 在宿主机独占物理核。

使用方法：

宿主机处理器架构为 x86，可以在宿主机的虚拟机 XML 中配置“hint-dedicated”和“host-passthrough”使能该特性，通过虚拟机 XML 配置将 vCPU 独占物理核的状态传递给虚拟机。vCPU 独占物理核的状态由宿主机保证。

```
<domain type='kvm'>
...
<features>
...
  <kvm>
    <hint-dedicated state='on'/>
  </kvm>
...
</features>
<cpu mode='host-passthrough' check='partial'/>
...
</domain>
```

确认 Guest-Idle-Haltpoll 特性是否生效。在虚拟机中执行如下命令，若返回 haltpoll，说明特性已经生效。

```
# cat /sys/devices/system/cpu/cpuidle/current_driver
```

配置 Guest-Idle-Haltpoll 参数（可选）。 虚拟机的 /sys/module/haltpoll/parameters/路径下提供了如下配置文件，用于调整配置参数，用户可以根据业务特点选择调整。

参数	说明

guest_halt_poll_ns	全局参数，指 vCPU 空闲后 polling 的最大时长，默认值为 200000（单位 ns）
guest_halt_poll_shrink	当唤醒事件发生在全局 guest_halt_poll_ns 时间之后，用于收缩当前 vCPU guest_halt_poll_ns 的除数因子，默认值为 2
guest_halt_poll_grow	当唤醒事件发生在当前 vCPU guest_halt_poll_ns 之后且在全局 guest_halt_poll_ns 之前，用于扩展当前 vCPU guest_halt_poll_ns 的乘数因子，默认值为 2
guest_halt_poll_grow_start	当系统空闲时，每个 vCPU 的 guest_halt_poll_ns 最终会达到零。该参数用于设置当前 vCPU guest_halt_poll_ns 的初始值，以便 vCPU polling 时长的收缩和扩展。默认值为 50000（单位 ns）
guest_halt_poll_allow_shrink	允许每个 vCPU guest_halt_poll_ns 收缩的开关，默认值是 Y（Y 表示允许收缩，N 表示禁止收缩）

可以使用 root 权限，参考如下命令修改参数值。其中 value 表示需要设置的参数值， configFile 为对应的配置文件。

```
# echo value > /sys/module/haltpoll/parameters/configFile
```

例如设置全局 guest_halt_poll_ns 为 200000ns 的命令如下：

```
# echo 200000 > /sys/module/haltpoll/parameters/guest_halt_poll_ns
```

4.2.7. 基于 MCE 定位受影响虚拟机

MCE 是 Intel 实现的一种机器检查架构，提供能够检测和报告硬件（机器）错误的机制。libvirt 支持获取系统 MCE 报告的内存错误故障点，然后通过对所有运行中的虚拟机进行内存映射检索，诊断受影响的具体虚拟机。该功能能够更精准地诊断与定位硬件故障对虚拟化业务的影响，便于进行故障处理，减少对虚拟化业务的影响和故障处理的开销。

注：MCE 只在 Intel 架构 CPU 上支持，其他架构的机器上不支持 MCE 机制，也就无法支持该功能。

使用方法：

功能依赖于 mcelog 软件包，mcelog 通过触发器的方式监听系统 MCE 输出信息。当出现 MCE 内存错误时，mcelog 将提取内存错误地址并传输给 libvirt。

安装与启动 mcelog 服务：

```
#yum install mcelog
#systemctl start mcelog
```

安装并启动 libvirtd 服务：

```
#yum install libvirt
#systemctl start libvirtd
```

如果已开启服务，可以在开启 mcelog 服务后重启 libvirtd 服务来开启功能：

```
#systemctl restart libvirtd
```

功能开启后，只有当手动进行 MCE 注错测试，以及系统 MCE 机制捕捉到实际发生的内存错误时，才会在 libvirtd 服务中输出相应的通知信息。

注： 未安装 mcelog 时，libvirt 将不会开启内存错误定位功能，并且在 libvirtd 服务中输出信息，提示未开启 mcelog 服务，不会开启功能。

4.3. 安全

4.3.1. 可信引导

4.3.1.1. vTPM 介绍

可信启动包含度量启动和远程证明。其中虚拟化组件主要提供度量启动功能。度量启动的两个基本要素是信任根和信任链，其基本思想是首先在计算机系统中建立一个信任根，从信任根开始到 BIOS/Bootloader、操作系统、再到应用，一级度量认证一级，一级信任一级，最终扩展到整个系统。启动过程中，前一个部件度量(计算 HASH 值)后一个部件，然后把度量值存入可信存储区，比如扩展到 TPM 的 PCR 中。后续接入平台后，部件的度量会上报到认证平台，认证平台会有配置一个可信白名单，如果某个部件的版本信息不在白名单里，则认为此设备是不受信任。

4.3.1.2. 使用方法

swtpm 提供了一个可集成到虚拟化环境中的 TPM 仿真器，已适配 qemu，swtpm 提供虚拟 TPM 设备(如 TPM1.2、TPM2.0)的模拟实现。

```
#yum install libtpms swtpm swtpm-devel swtpm-tools
```

虚拟机配置 vTPM 设备

```
<devices>
  ...
  <tpm model='tpm-device'>
    <backend type='emulator' version='2.0'/>
  </tpm>
  ...
</devices>
```

度量启动功能使能与否由 vBIOS 决定,目前的 vBIOS 已经具备了度量启动的能力。若宿主机采用其他版本的 edk2 组件，请确认是否支持度量启动功能。

使用 root 用户登录虚拟机，确认虚拟机中是否安装了 tpm 驱动，tpm2-tss 以及 tpm2-tools 工具。

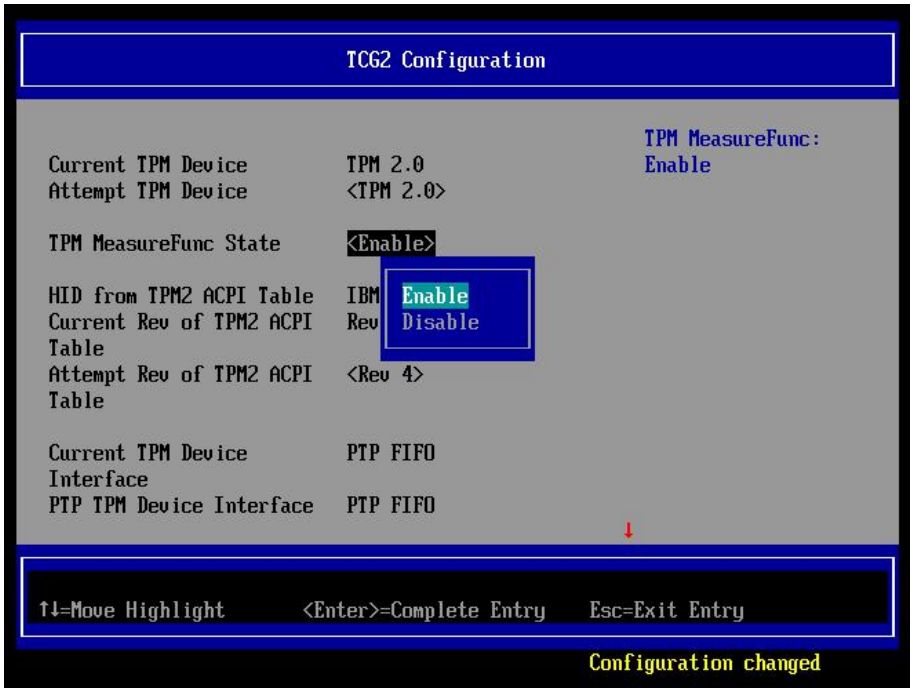
```
# lsmod |grep tpm
# tpm_tis          16384   0
# yum list installed | grep -E 'tpm2-tss|tpm2-tools'
# yum install tpm2-tss tpm2-tools
```

在虚拟机中使用 tpm2_pcrread 读取所有的 pcr 值，目前已支持国密 sm3 算法，用户可以通过修改/usr/bin/swtpm_setup.sh 文件中默认加密算法来启用 sm3 算法

```
# tpm2_pcrread
sm3 :
  0  : fffdcae7cef57d93c5f64d1f9b7f1879275cff55
  1  : 5387ba1d17bba5fdadb77621376250c2396c5413
  2  : b2a83b0ebf2f8374299a5b2bdfc31ea955ad7236
```

5.3.1.3 vtpm 菜单控制

edk2 提供添加 vtpm 设备后对可信度量的管理功能,用户可以决定在 vtpm 添加后是否启用对系统度量功能



4.3.2. 磁盘国密加密

虚拟机磁盘国密加密是 qemu 基于 nettle 加密算法库实现的国密 SM4 加密算法，并提出对虚拟机实现加密的创建流程。虚拟机磁盘国密加密是在虚拟机已经创建完成的基础上，将虚拟机磁盘转换成加密磁盘，并为虚拟机配置相应的密码。使得加密后的磁盘只能在当前物理机上使用。使用方法如下：

1. 创建含有密码的加密磁盘。

```
    # MYSECRET=`printf %s "123456" | base64`
    #qemu-img create -f luks --object
secret,data=$MYSECRET,id=sec0,format=base64,qom-type=secret -o
key-secret=sec0,cipher-alg=sm4 encrypt.luks 20G
    # qemu-img info encrypt.luks
    image: encrypt.luks
    file format: luks
    virtual size: 20 GiB (21474836480 bytes)
    disk size: 132 KiB
    encrypted: yes
    Format specific information:
       ivgen alg: plain64
       hash alg: sha256
       cipher alg: sm4
       uuid: f60f362d-441d-4afc-a552-b62f279c35f7
       cipher mode: xts8
       slots:
          [0]:
             active: true
             iters: 3604838
             key offset: 4096
             stripes: 4000
          [1]:
             active: false
             key offset: 135168
          [2]:
             active: false
             key offset: 266240
          [3]:
             active: false
             key offset: 397312
          [4]:
             active: false
             key offset: 528384
          [5]:
             active: false
              key offset: 659456
          [6]:
             active: false
             key offset: 790528
          [7]:
             active: false
             key offset: 921600
       payload offset: 1052672
       master key iters: 446187
```

1. 将已经装好操作系统由普通磁盘转换到加密磁盘中

```
# qemu-img convert --target-image-opts \
--object secret,data=$MYSECRET,id=sec0 -f qcow2 demo.qcow2 \
-n driver=luks,file.filename=encrypt.luks,key-secret=sec0
```

2. 创建 xml 秘钥文件，其中 uuid 为加密磁盘的 uuid 信息，见第一步。创建之后，调用 virsh 的密码定义指令，将磁盘 uuid 存入到物理机操作系统中；最后为磁盘的 uuid 设置密码，且密码需与加密磁盘的密码一致。

```
# vi sec.xml
<secret ephemeral='no' private='yes'>
 <uuid>f60f362d-441d-4afc-a552-b62f279c35f7</uuid>
</secret>
# virsh secret-define sec.xml
Secret f60f362d-441d-4afc-a552-b62f279c35f7 created
# virsh secret-set-value 4bcfa5a0-b8ad-497f-a8f7-4dcf774d22c8
$MYSECRET
```

3. 配置原虚拟机 xml 文件，将磁盘 uuid 写入到 xml 文件中

```
    <devices>
      <emulator>/usr/libexec/qemu-kvm</emulator>
      <disk type='file' device='disk'>
        <driver name='qemu' type='raw'/>
        <source file='/var/lib/libvirt/images/encrypt.luks'/>
        <target dev='vda' bus='virtio'/>
        <encryption format='luks'>
          <secret type='passphrase'
uuid='f60f362d-441d-4afc-a552-b62f279c35f7' />
        </encryption>
        <address type='pci' domain='0x0000' bus='0x05' slot='0x00'
function='0x0'/>
      </disk>
```

4.3.3. 国密热迁移

虚拟机热迁移国密加密是 qemu 基于 gnutls 证书管理工具以及 nettle 加密算法库实现的国密加密功能，是对现有的虚拟机热迁移加密的升级和补充。新加入的特性包括使用国密 SM3 加密证书对客户端的身份认证，以及开启国密 SM4 加密算法的 TLS 通道。使用方法如下：

1. 修改迁移服务器和客户端的 hostname。以下操作都需要在多平台操作

```
# hostnamectl set-hostname xxx
# su
```

2. 创建客户端和服务端物理机操作系统秘钥，并相互发送建立互信。

```
# ssh-keygen
# ssh-copy-id root@ip
```

3. 在客户端及服务端创建证书存放文件

```
# mkdir -p /etc/pki/qemu
# mkdir -p /etc/pki/libvirt/private
# mkdir -p /etc/pki/CA
```

4. 配置一级秘钥的证书模板。其中证书组织（cn）可以修改

```
# vi ca.info
cn = Kylin
ca
cert_signing_key
expiration_days = 700
```

5. 生成一级秘钥，并创建 CA 证书

```
# certtool --generate-privkey --sm2 > cakey.pem
Generating a 256 bit SM2 private key ... # certtool --generate-self-signed
--load-privkey cakey.pem
--template ca.info --outfile cacert.pem
Generating a self signed certificate... X.509 Certificate Information:
Version: 3
Serial Number (hex):
56e12f2eb019af4c658fe337edb16c54642235e6
Validity:
Not Before: Thu Nov 10 01:18:24 UTC 2022
Not After: Thu Oct 10 01:18:24 UTC 2024
Subject: CN=Kylin \
Subject Public Key Algorithm: SM2
Algorithm Security Level: High (256 bits)
.....
```

6. 下派证书到服务器以及客户端。user 及 ip 对应目标物理机的信息

```
# scp cacert.pem user@ip:/etc/pki/CA
```

7. 创建服务端的证书模板；创建服务机的私钥。其中 organizationd 的值要与 CA 证书的值一致。cn 的值要与当前物理机的 hostname 保持一致。

```
# vi host1-server.info
organization = Kylin
cn = host1
tls_www_server
encryption_key
signing_key
# certtool --generate-privkey --sm2> host1-serverkey.pem
Generating a 256 bit SM2 private key ...
```

8. 创建服务器私钥证书，并下放到服务端的指定文件夹中

```
    # certtool --generate-certificate --template host1-server.info \
--load-privkey hos 赋予权限 t1-serverkey.pem \ --load-ca-certificate
cacert.pem \ --load-ca-privkey cakey.pem \ --outfile host1-servercert.pem
    Generating a signed certificate... X.509 Certificate Information:
    Version: 3
    Serial Number (hex):
    3f015b3eb06154da4be3b4a4bf4c5cb04554803e
    Validity:
    Not Before: Thu Nov 10 02:00:00 UTC 2022
    Not After: Fri Nov 10 02:00:00 UTC 2023
    Subject: O=Kylin \ ,CN=sub130 \
    Subject Public Key Algorithm: SM2
    Algorithm Security Level: High (256 bits)
    # scp host1-servercert.pem
    user@ip:/etc/pki/libvirt/servercert.pem
    # scp host1-serverkey.pem
    user@ip:/etc/pki/libvirt/private/serverkey.pem
```

9. 创建客户端证书模板，以及客户端私钥。其中 organizationd 的值要与 CA 证书的值一致。cn 的值要与当前物理机的 hostname 保持一致。

```
# vi host2-client.info
country = CN
state = CN
locality = TJ
organization = Kylin
cn = host2
tls_www_client
encryption_key
signing_key
# certtool --generate-privkey --sm2> host2-clientkey.pem
Generating a 256 bit SM2 private key ...
```

10. 创建客户端证书私钥，并下放到客户端的指定文件夹中，并固定文件名。

```
    # certtool --generate-certificate \ --template host2-client.info \
--load-privkey host2-clientkey.pem \ --load-ca-certificate cacert.pem \
--load-ca-privkey cakey.pem \ --outfile host2-clientcert.pem
    # scp host2-clientcert.pem
    user@ip:/etc/pki/libvirt/clientcert.pem
    # scp host2-clientkey.pem
    user@ip:/etc/pki/libvirt/private/clientkey.pem
```

11. 进入服务器与客户端中，配置客户端与服务端建立连接时证书权限

```
    # chown root:qemu /etc/pki/libvirt
    # chown root:qemu /etc/pki/libvirt/*
    # chown root:qemu /etc/pki/libvirt/private
    # chmod 440 /etc/pki/libvirt/servercert.pem
    # chmod 750 /etc/pki/libvirt/private/
    # chmod 600 /etc/pki/libvirt/private/serverkey.pem
    # chmod 400 /etc/pki/libvirt/clientcert.pem
    # chmod 644 /etc/pki/libvirt/private/clientkey.pem
```

12. 在服务器端与客户端中复制所有的 CA 证书到迁移区域文件夹 /etc/pki/libvirt-migrate 中，并修改文件权限

```
# cp -a /etc/pki/CA/cacert.pem
/etc/pki/libvirt-migrate/ca-cert.pem
# cp -a /etc/pki/libvirt/servercert.pem
/etc/pki/libvirt-migrate/server-cert.pem
# cp -a /etc/pki/libvirt/private/serverkey.pem
/etc/pki/libvirt-migrate/server-key.pem
# cp -a /etc/pki/libvirt/clientcert.pem
/etc/pki/libvirt-migrate/client-cert.pem
# cp -a /etc/pki/libvirt/private/clientkey.pem
/etc/pki/libvirt-migrate/client-key.pem
# chown qemu:qemu /etc/pki/libvirt-migrate/server-cert.pem
# chown qemu:qemu /etc/pki/libvirt-migrate/server-key.pem
# chown qemu:qemu /etc/pki/libvirt-migrate/client-cert.pem
# chown qemu:qemu /etc/pki/libvirt-migrate/client-key.pem
```

13. 修改热迁移加密的配置文件，释放对应参数的注释。

```
# vi /etc/libvirt/qemu/qemu.conf
migrate_tls_x509_cert_dir
migrate_tls_x509_verify
migrate_tls_x509_verify
```

14. 配置 libvirt 监听接口，并重启 libvirtd 服务和关闭防火墙

```
# vi /etc/libvirt/libvirtd.conf
tls_port="16514" # systemctl restart libvirtd
# systemctl stop firewalld
```

15. 执行迁移命令

```
# virsh migrate --live --unsafe --tls --verbose --domain xxx
--desturi qemu+ssh://hostname/system
```

4.3.4. 远程桌面国密保护

当前主流的虚拟机远程桌面有 vnc、spice 两种，下面分别介绍创建两种远程桌面的国密算法加密通道。

spice 国密加密通道

1. 环境配置

查看是否安装 openssl 安装包。

```
rpm -qa | grep openssl
```

2. 在远程桌面服务端创建证书：

创建根证书密钥参数文件

```
openssl ecparam -out sm2param.pem -name SM2
```

创建根证书和根证书密钥，执行过程需要输入一个密钥口令，引号中 C 指定国家，L 指定地区或城市，O 指定组织，CN 指定域名

```
openssl req -x509 -newkey param:sm2param.pem -keyout ca-key.pem
-out ca-cert.pem -sm3   -subj "/C=CN/L=TJ/O=kylin/CN=my CA"
```

生成服务端私钥

```
SERVER_KEY=server-key.pem
openssl genrsa -out $SERVER_KEY 1024
```

生成签发证书请求文件

```
openssl req -new -key $SERVER_KEY -out server-key.csr -utf8 -subj
"/C=CN/L=TJ/O=kylin/CN=my server"
```

签发服务端证书，签发过程中需要输入根证书密钥口令

```
openssl x509 -req -days 1095 -in server-key.csr -CA ca-cert.pem -CAkey
ca-key.pem -set_serial 01 -out server-cert.pem
```

现在创建一个无需口令的密钥

```
openssl rsa -in $SERVER_KEY -out $SERVER_KEY.insecure
mv $SERVER_KEY $SERVER_KEY.secure
mv $SERVER_KEY.insecure $SERVER_KEY
```

3. 查看证书文件

```
# ls
ca-cert.pem ca-key.pem server-cert.pem server-key.csr server-key.pem
server-key.pem.secure spice_cert.sh
```

将上述生成的证书文件拷贝到服务端的/etc/pki/libvirt-spice 文件夹中。

```
cp ./*.pem /etc/pki/libvirt-spice
```

4. 在宿主机上修改 libvirtd 服务的配置文件并加载证书

```
#vim /etc/libvirt/qemu.conf
```

去掉以下三项的注释

```
#指点 spice server 的监听地址
spice_listen="0.0.0.0"
#使能 tls 加密通道
spice_tls=1
#指定加密证书文件路径
spice_tls_x509_cert_dir="/etc/pki/libvirt-spice"
```

重启 libvritd 加载证书

```
#systemctl deamon-reload
#systemctl restart libvirtd
```

5. 通过 virsh edit 配置虚拟机使用国密加密的远程桌面，以下为示例

```
        <graphics type='spice' tlsPort='5999' autoport='no' listen='ip
    address'>
          <listen type='address' address='ip address'/>
          <channel name='main' mode='secure'/>
          <image compression='off'/>
          <gl enable='no'/>
        </graphics>
```

用户可以自定义远程桌面服务使用的宿主机端口，监听 ip 地址和 spice 通道。

6. linux 客户端可以启动 spicy 访问，客户端只需要 ca-cert.pem，用户便可以通过 TLS 通道访问虚拟机桌面，

检查客户端是否安装：

```
rpm -qa spice-gtk
```

访问服务端示例：

```
spicy --spice-ca-file=/etc/pki/libvirt-spice/ca-cert.pem
spice://<hostname>?tls-port=5999 --spice-host-subject="C=CN, L=TJ,
O=Kylin, CN=my_server"
```

vnc 国密加密通道

先决条件：服务端客户端之间可以互相 ssh 免密登录。

1. 服务端创建 vnc-cert.sh 文件，用户可自定义 CN 域名。

以下为示例，host1，host2 均可为服务端客户端：

```
certtool --generate-privkey --sm2 > ca-key.pem

cat > ca.info<<EOF
cn = Kylin
ca
cert_signing_key
EOF

certtool --generate-self-signed \
    --load-privkey ca-key.pem \
    --template ca.info \
    --outfile ca-cert.pem

cat > server1.info<<EOF
organization = Kylin
cn = host1
tls_www_server
encryption_key
signing_key
EOF
cat > server2.info<<EOF
organization = Kylin
cn = host2
tls_www_server
encryption_key
signing_key
EOF

certtool --generate-privkey --sm2> server1-key.pem
certtool --generate-certificate \
    --load-ca-certificate ca-cert.pem \
    --load-ca-privkey ca-key.pem \
    --load-privkey server1-key.pem \
    --template server1.info \
    --outfile server1-cert.pem

certtool --generate-privkey --sm2> server2-key.pem
certtool --generate-certificate \
    --load-ca-certificate ca-cert.pem \
    --load-ca-privkey ca-key.pem \
    --load-privkey server2-key.pem \
    --template server2.info \
    --outfile server2-cert.pem

cat > client1.info<<EOF
organization = Kylin
cn = host1
tls_www_client
encryption_key
signing_key
EOF
```

```
cat > client2.info<<EOF
organization = Kylin
cn = host2
tls_www_client
encryption_key
signing_key
EOF


certtool --generate-privkey --sm2> client1-key.pem
certtool --generate-certificate \
    --load-ca-certificate ca-cert.pem \
    --load-ca-privkey ca-key.pem \
    --load-privkey client1-key.pem \
    --template client1.info \
    --outfile client1-cert.pem

certtool --generate-privkey --sm2> client2-key.pem
certtool --generate-certificate \
    --load-ca-certificate ca-cert.pem \
    --load-ca-privkey ca-key.pem \
    --load-privkey client2-key.pem \
    --template client2.info \
    --outfile client2-cert.pem

rm -rf /etc/pki/libvirt-vnc/*
\cp server1-cert.pem /etc/pki/libvirt-vnc/server-cert.pem
\cp server1-key.pem /etc/pki/libvirt-vnc/server-key.pem
\cp ca-cert.pem /etc/pki/libvirt-vnc/
\cp server1-cert.pem /etc/pki/libvirt/servercert.pem
\cp server1-key.pem /etc/pki/libvirt/private/serverkey.pem
\cp ca-cert.pem /etc/pki/CA/cacert.pem


scp client2-cert.pem root@host2:/etc/pki/libvirt/clientcert.pem
scp client2-key.pem root@host2:/etc/pki/libvirt/private/clientkey.pem
scp ca-cert.pem root@host2:/etc/pki/CA/cacert.pem

sudo chgrp qemu /etc/pki/libvirt \
        /etc/pki/libvirt-vnc \
        /etc/pki/libvirt/private \
        /etc/pki/libvirt/servercert.pem \
        /etc/pki/libvirt/private/serverkey.pem

sudo chmod 750 /etc/pki/libvirt \
        /etc/pki/libvirt-vnc \
        /etc/pki/libvirt/private

sudo chmod 440 /etc/pki/libvirt/servercert.pem \
        /etc/pki/libvirt/private/serverkey.pem
```

2. 在宿主机上修改 libvirtd 服务的配置文件并加载证书

```
#vim /etc/libvirt/qemu.conf
```

去掉下面这四项的注释

```
vnc_listen="0.0.0.0"
vnc_tls=1
vnc_tls_x509_cert_dir="/etc/pki/libvirt-vnc"
vnc_tls_x509_verify = 1 #配置为 1，表示 TLS 认证使用 X509 证书
```

启动 libvirtd 服务

```
/usr/sbin/libvirtd  --listen -d
```

3. 虚拟机配置

以下为配置示例：

```
<graphics type='vnc' port='5904' '>
 <listen type='address' address='ip address'/>
</graphics>
```

4. 客户端连接 vnc 虚拟机

```
virt-viewer -c qemu+tls://root@host1 或 host2/system guestname
```

4.4. io_uring

4.4.1. iouring 原理介绍

io_uring 的原理是让用户态进程与内核通过一个共享内存的无锁环形队列进行高效交互。相关的技术原理其实与 DPDK/SPDK 中的 rte_ring 以及 virtio 的 vring 是差不多的，只是这些技术不涉及用户态和内核态的共享内存。高性能网络 IO 框架 netmap 与 io_uring 技术原理更加接近，都是通过共享内存和无锁队列技术实现用户态和内核态高效交互。为了最大程度的减少系统调用过程中的参数内存拷贝，io_uring 采用了将内核态地址空间映射到用户态的方式。通过在用户态对 io_uring fd 进行 mmap，可以获得 io_uring 相关的两个内核队列（IO 请求和 IO 完成事件）的用户态地址。用户态程序可以直接操作这两个队列来向内核发送 IO 请求，接收内核完成 IO 的事件通知。IO 请求和完成事件不需要通过系统调用传递，也就完全避免了 copy_to_user/copy_from_user 的开销。

io_uring 使用了单生产者单消费者的无锁队列来实现用户态程序与内核对共享内存的高效并发访问，生产者只修改队尾指针，消费者只修改队头指针，不会互相阻塞。对于 IO 请求队列来说，用户态程序是生产者内核是消费者，完成事件队列则相反。需要注意的是由于队列是单生产者单消费者的，因此如果用户态程序需要并发访问队列，需要自己保证一致性（锁/CAS）。

4.4.2. iouring 使用

以虚拟机磁盘为 raw 格式为例，修改虚拟机 xml 文件

```
<disk type='file' device='disk'>
    <driver name='qemu' type='raw' cache='none' io='io_uring'/>
    <source file='/virt-data/images/test.img'/>
    <target dev='sda' bus='scsi'/>
</disk>
```

重新定义域

```
virsh define test
```

启动虚拟机

```
virsh start test
```

4.5. 虚拟机可用性探测

QAPI 新增接口：query-virtio-blk，query-virtio-net。其功能用于查询指定虚拟机的 virtio-blk 或 virtio-net 的 VRing 状态信息。通过这一功能可以为用户提供 virtio 设备正在处理的当前 IO 队列状态，通过接口反馈的参数信息可判断，virtio 队列工作是否正常，后端 IO 处理引擎是否阻塞。侧面的为用户提供虚拟机 IO SLI 指标和参考依据。

查询 virtio-blk 的 VRing 状态信息，可执行如下两条指令（任选其一即可）：

```
#virsh qemu-monitor-command <domain> --hmp
query-virtio-blk;
#virsh qemu-monitor-command <domain> [--pretty] \
"{\"execute\":\"query-virtio-blk\"}";
```

执行结果如下：

```
[root@hostos]# virsh qemu-monitor-command wlj-80 --hmp
query-virtio-blk;
Device id: "virtio-disk0" vq_desc_head: 2
vq_avail_idx: 54
vq_used_idx: 54
vq_inuse: 0
vq_last_num_reqs: 2
[root@hostos]# virsh qemu-monitor-command wlj-80 \
> "{\"execute\":\"query-virtio-blk\"}" --pretty;
{
"return": [
{
"vq-inuse": 0, "device-id": "virtio-disk0", "vq-used-idx": 114,
"vq-last-num-reqs": 1, "vq-desc-head": 36, "vq-avail-idx": 114
}
],"id": "libvirt-23" }
```

参数	描述
device-id	virtio-blk 设备 ID
vq-avail-idx	Guest 上一次增加的 buffer 在 Avail Ring 中的位置
vq-used-idx	Host 工作在 Used Ring 中的位置
vq-desc-head	Desc Ring 中 head buffer 在 Desc Table 的索引
vq-last-num-reqs	上一次发送数据提交的 req 数量

查询 virtio-net 的 VRing 状态信息，可执行如下两条指令（任选其一即可）：

```
#virsh qemu-monitor-command <domain> --hmp
query-virtio-net;
#virsh qemu-monitor-command <domain> [--pretty] \
"{\"execute\":\"query-virtio-net\"}";
```

执行结果如下：

```
[root@hostos]# virsh qemu-monitor-command wlj-80 --hmp
query-virtio-net;
Device id: "net0" vring_idx: 0
rx_desc_head: 21
rx_avail_idx: 18
rx_used_idx: 41
rx_inuse: 0
tx_desc_head: 1
tx_avail_idx: 139
tx_used_idx: 139
tx_inuse: 0
[root@hostos]# virsh qemu-monitor-command wlj-80 \
> "{\"execute\":\"query-virtio-net\"}" --pretty;
{
"return": [
{
"net-device-id": "net0", "vrings-info": [
{
"rx-inuse": 0, "tx-avail-idx": 139, "rx-used-idx": 135, "tx-inuse": 0,
"rx-desc-head": 21, "tx-used-idx": 139, "rx-avail-idx": 18, "vring-idx": 0,
"tx-desc-head": 1
}
        ]
      }
   ],
   "id": "libvirt-25"
}
```

参数	描述

net-device-id	virtio-net 设备 ID
rx-avail-idx	Guest 上一次增加的 buffer 在 rx Avail Ring 中的位置
rx-used-idx	Host 工作在 rx Used Ring 中的位置
rx-desc-head	rx Desc Ring 中 head buffer 在 Desc Table 的索引
rx-inuse	rx 队列 vring 已使用数量
tx-avail-idx	Guest 上一次增加的 buffer 在 tx Avail Ring 中的位置
tx-used-idx	Host 工作在 tx Used Ring 中的位置
tx-desc-head	tx Desc Ring 中 head buffer 在 Desc Table 的索引
tx-inuse	tx 队列 vring 已使用数量

虚拟机可用性检测：

1. 当检测指令（query-virtio-blk/net）未收到检测结果，即检测超时，则可判断虚拟机当前不可用；

2. 当向虚拟机周期性发出检测指令时，前后指令收到的检测结果中数据未发生变化，可判定虚拟机当前状态异常（注：周期间隔及数据变化阈值，可根据具体业务场景自行调配）。

4.6. virtiofs 使用

virtiofs 文件系统实现了一个半虚拟化 VIRTIO 类型“virtio-fs”设备的驱动，通过该类型设备实现客机<->主机文件系统共享。它允许客机挂载一个已经导出到主机的目录。使用 virtiofs 在主机和虚拟机间共享文件：

先决条件：开启虚拟化，共享文件的虚拟机操作系统为 linux 发行版。

存在被共享的文件夹，不存在可以创建：

```
#mkdir <path_to_directory>
```

1. 在主机和虚拟机间共享文件

检查软件包

```
#rpm -qa libvirt  #检查是否存在 libvrt
#rpm -qa qemu     #检查是否存在 qemu
```

修改需要共享文件的虚拟机的 xml 配置

打开虚拟机的配置

```
#virsh edit <vm-name>
```

在虚拟机 xml 的<device>部分添加以下内容

```
<filesystem type='mount' accessmode='passthrough'>
  <driver type='virtiofs'/>
  <binary path='/usr/libexec/virtiofsd' xattr='on'/>
  <source dir='path_to_directory'/>
  <target dir='mount_tag'/>
</filesystem>
```

为 xml 配置共享内存的 NUMA 拓扑，以下示例为所有 CPU 和所有 RAM 添加基本拓扑

```
<cpu mode='host-passthrough' check='none'>
  <numa>
    <cell id='0' cpus='0-{number-vcpus - 1}' memory='{ram-amount-KiB}' unit='KiB' memAccess='shared'/>
  </numa>
</cpu>
```

将共享内存支持添加到 xml 配置的<domain>部分

```
<domain>
 [...]
 <memoryBacking>
   <access mode='shared'/>
 </memoryBacking>
 [...]
</domain>
```

引导虚拟机

```
#virsh start <vm-name>
```

在客户端操作系统中挂载文件系统，示例将 virtiofs 文件系统挂载到/mnt

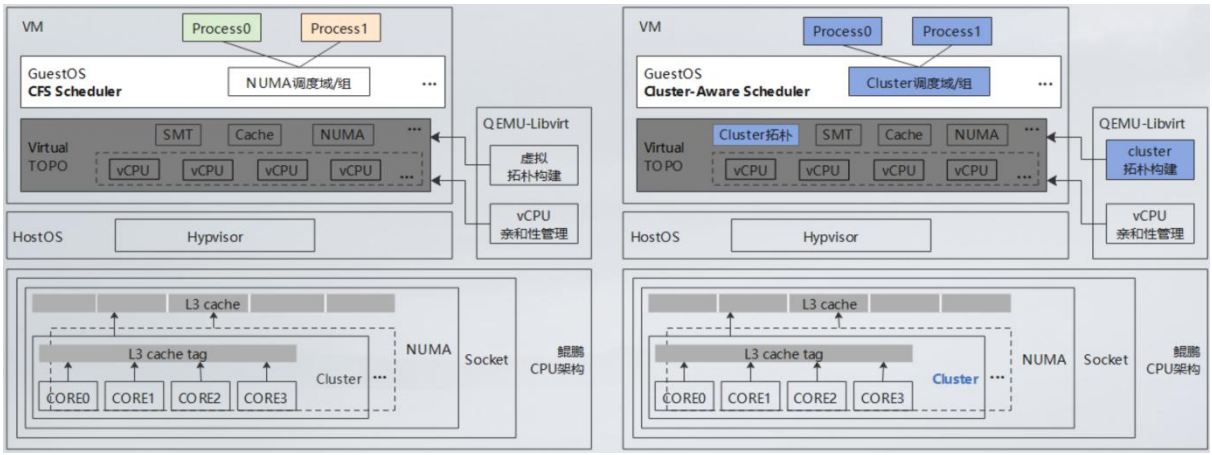
```
#mount -t virtiofs <mount_tag> /mnt
```

4.7. 鲲鹏 cluster 使能

4.7.1. 简介

集群调度器的支持，对于具有 CPU 内核集群的系统，可用于改进 CPU 调度器行为，减少任务在 CPU 上的切换时间。

ARM64 服务器芯片（鲲鹏 920）在每个 NUMA 节点上有 6 个或 8 个 cluster，每个 cluster 有 4 个 cpu。所有 cluster 共享 L3 缓存数据，而每个 cluster 都有本地 L3 tag。另一方面，每个 cluster 将共享一些内部系统总线。这意味着缓存在一个 cluster 内的亲和性要比跨 cluster 强得多。



将 Cluster-aware 用于虚拟机环境中，有利于虚拟机能够更好的适用和利用 CPU 硬件和集群资源,通过将 Cluster 拓扑信息透传给虚拟机,在 GuestOS 中提前分配 CPU 资源，可以极大的提高虚拟机的运行性能。

4.7.2. 特性使能

1. 若要使能集群调度器，需要在构建 kernel 时设置“CONFIG_SCHED_CLUSTER”选项，用以增强 CPU 调度器基于内核布局的决策过程。

检查当前内核是否开启了集群调度器支持，需确认相关配置项已开启:

```
CONFIG_SCHED_CLUSTER=y
```

2. 对虚拟机的 xml 文件进行配置

```
<cpu mode='host-passthrough' check='none'>
  <topology sockets='1' dies='1' clusters='2' cores='4' threads='1'/>
</cpu>
```

当前含义是 8 个 cpu 分成两个簇，每个簇里有 4 个核心。一共使用了 8 个 cpu

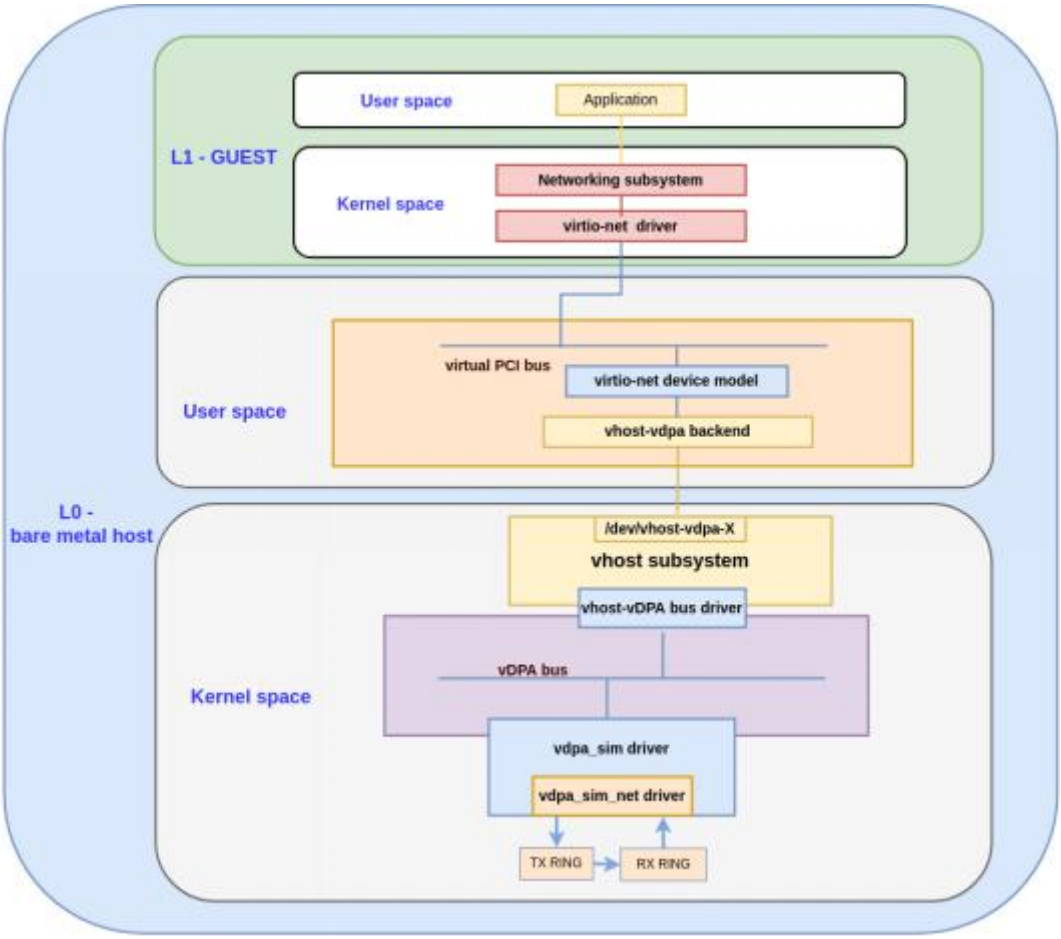
虚拟机显示为：

```
cat /sys/devices/system/cpu/cpu0/topology/cluster_cpus_list
0-3
```

4.8. VDPA

4.8.1. vdpa 简介

vdpa: Vhost Data Path Acceleration，意为 vhost 数据路径加速，它支持 virtio ring 兼容设备，可以直接为 virtio 驱动程序提供数据通路加速功能。vDPA 内核框架的主要目标是隐藏 vDPA hardware 实现的复杂性，并为内核和用户空间提供一个安全统一的接口来使用。从硬件的角度来看，vDPA 可以由多种不同类型的设备实现。



对于用户空间驱动程序，vDPA 框架呈现是一个 vhost 字符设备，允许用户空间 vhost 驱动程序控制 vdpa 设备，如同 vhost-device 设备。各种用户空间应用程序可以通过该方式连接到 vhost device。可以将 vDPA 设备与 Qemu 连接起来，并将其用作运行在 guest 内部的 virtio 驱动程序的 vhost 后端。此时可将 vDPA device 看做是 vhost-net 之外的另一种类型的数据面卸载的目标。对于内核 virtio 驱动程序，vDPA 框架将提供一个 virtio 设备，允许内核 virtio 驱动程序像控制标准 virtio 设备来控制 vDPA 设备。vDPA 框架是一个独立的子系统，为了灵活地实现新的硬件功能。为了支持热迁移，该框架支持通过现有的 vhost API 保存和恢复设备状态，可以很容易地扩展到支持硬件脏页跟踪或软件辅助脏页跟踪。

4.8.2. vdpa 使用

确认 vdpa 相关配置项已开启并安装相关驱动模块：

```
CONFIG_VHOST_IOTLB=m
CONFIG_VHOST=m
CONFIG_VDPA=m
CONFIG_VHOST_VDPA=m
CONFIG_VDPA_SIM=m
CONFIG_VDPA_SIM_NET=m
```

安装相关驱动：

```
modprobe vdpa_sim_net
modprobe vhost_vdpa
```

验证 vdpa

创建 vdpa 设备：

```
vdpa dev add name vdpa0 mgmtdev vdpasim_net
```

```
[root@localhost ~]# vdpa mgmtdev show
vdpasim_net:
  supported_classes net
[root@localhost ~]#
```

```
[root@localhost ~]# vdpa dev add name vdpa0 mgmtdev vdpasim_net
[root@localhost ~]# vdpa dev
vdpa0: type network mgmtdev vdpasim_net vendor_id 0 max_vqs 2 max_vq_size 256
```

```
[root@localhost software]# ls /dev/ |grep vdpa
vhost-vdpa-0
[root@localhost software]#
```

使用 vdpa 设备(因 libvirt 暂未支持 xml 中添加 vdpa 设备，可在 qemu 命令行中末尾添加如下参数)：

```
qemu-kvm *** vhost-vdpa-device-pci,vhostdev=/dev/vhost-vdpa-0
```

验证可用性:

虚拟机内查看网卡信息，确认网卡驱动为 virtio_net 即可

```
[root@localhost ~]# ethtool -i ens2
driver: virtio_net
version: 1.0.0
firmware-version:
expansion-rom-version:
bus-info: 0000:00:02.0
supports-statistics: yes
supports-test: no
supports-eeprom-access: no
supports-register-dump: no
supports-priv-flags: no
[root@localhost ~]#
```

4.9. TLBI 广播优化

4.9.1. 特性介绍

在虚拟化场景，硬件不感知一个虚拟机 vCPU 线程具体在哪些 pCPU 上运行过，因此 vCPU 的 TLBI 指令会被统一广播到全系统范围的所有物理核上，并在所有物理核上刷新 TLB，这里会产生不必要的性能开销。从功能上分析，硬件只需要在该 vCPU

运行过的那些 pCPU 上刷新 TLB 即可

DVMBM 特性，用于 TLBI 广播优化。新增寄存器用于检测硬件是否支持该特性、开启/关闭特性、配置 TLBI 的广播范围等。开启 TLBI 广播优化特性后，在虚拟化场景，通过配置一个 pCPU 的 LSUDVMBM_EL2 寄存器，可以控制该 pCPU 上的虚拟机 TLBI 指令只在指定的 CPU 范围内广播，从而避免不必要的开销。

新增内核启动参数，可以在用户态全局地开启/关闭 DVMBM 特性，系统默认为关闭。若要使能该特性，可以在 Host 内核的启动命令中配置该参数值为 1，即 kvm-arm.dvmbm_enabled=1，在 KVM 初始化阶段，若系统检测到硬件支持 DVMBM 特性且参数 dvmbm_enabled 的值为 1 时，会将所有的物理核的 DVMBM 特性开启，并打印 kvm [1]: KVM dvmbm enabled 。否则，会将所有的物理核的 DVMBM 特性关闭，并打印 kvm [1]: KVM dvmbm disabled 。

```
CONFIG_KVM_HISI_VIRT
```

4.9.2. 特性应用

LSUDVMBM_EL2 寄存器

增加 LSUDVMBM_EL2 寄存器，该寄存器用于保存 TLBI 范围广播的 CPU 位图。当虚拟机 vCPU 执行 TLBI 指令时，其所在的物理 CPU 会首先读取 LSUDVMBM_EL2 寄存器中保存的 CPU 位图，并只将 TLBI 指令广播至位图中指定的 pCPU 上，即只在一定范围内的物理 CPU 上刷新 TLB。

目前 TLBI 的广播范围以 VM 为粒度，即广播的 CPU 范围是该虚拟机所有 vCPU 绑核范围的并集。举例来说，配置 4U 虚拟机，vCPU 0-3 分别一一绑核至物理机的 2，4，6，8 号 CPU 上，那么该虚拟机的 TLBI 广播范围为[2, 4, 6, 8]。当虚拟机的某个 vCPU 执行 TLBI 指令时，该 vCPU 所运行的物理核会将 TLBI 指令统一广播至 2，4，6，8 号 CPU 上执行。

4.10. 虚拟机热迁移

4.10.1. 跨平台 v2v 迁移

V2V 迁移（virtual to virtual migration ）指将操作系统、应用程序和数据从现有的虚拟机或磁盘分区迁移到另一个虚拟机或磁盘分区，目标可以是一个或多个虚拟化平台。使用专用的迁移工具进行部分或完全的自动化迁移。

执行 V2V 迁移有助于在短时间内将一个虚拟机的内容，从一个虚拟化平台移动到另一个虚拟平台上，它节省了时间，减少了工作量，同时也减少了硬件和存储的成本。

virt-v2v 命令

我们目前常用的 V2V 迁移工具是 virt-v2v，它主要由 libguestfs 软件提供，并可以在 libguestfs 官方网站(https://libguestfs.org/virt-v2v.1.html) 中获取到更多详细信息。

virt-v2v 主要从不同的外部虚拟化平台上，将虚拟机整体转化到可以运行的 KVM 平台上。它可以读取在 VMware、Xen、Hyper-V 和其他虚拟机管理程序上运行的 Windows 和 Linux 的虚拟机，并将其转换为 KVM 的 libvirt、OpenStack、oVirt 等虚拟化平台上可用的虚拟机。

使用方法

```
virt-v2v [options]
```

在命令选项中，我们通常使用几个 -i * 选项来控制输入模式，同时使用几个 -o * 选项来控制输出模式。在这个意义上，“输入”指的是源外部虚拟机管理程序，如 VMware；而“输出”指的是基于 KVM 的目标管理系统，如 oVirt 或 OpenStack。

```
virt-v2v -i * -o*
```

常用的输入选项：

- -i disk：指定输入的设备为本地磁盘映像（例如 qcow2、raw 文件）
- -i ova：指定输入的设备为 ova 磁盘映像（由 VMware 或 oVirt 导出）
- -i libvirt：通过 libvirt 远程链接运行在其他虚拟化平台上的虚拟机

常用的输出选项：

- null：guest 被转换和复制，但是结果被丢弃并且没有元数据被写入；
- local：将转换后的虚拟机的磁盘映像与 libvirt xml 文件写入到-os dir/选项指定的本地目录下；
- qemu：类似于-o local，只是多了一个 shell 脚本，用来将输出的虚拟机磁盘映像引导在 qemu 中创建虚拟机。转换后的虚拟机的磁盘映像与 shell 文件写入到-os dir/选项指定的本地目录下；
- libvirt：这是默认值。在此模式下，转换后的 guest 虚拟机将被创建为 libvirt guest 虚拟机。您可以将虚拟机创建在本地，也可以通过-oc 选项指定远程 libvirt 主机作为输出端。
- glance：这是一个遗留选项。您可能应该使用-o openstack 来代替。 将输出方法设置为 OpenStack Glance。在这种模式下，转换后的客户机被上传到 Glance。
- rhv（与 rhev/ovirt 选项相同）：将输出方法设置为 rhv。转换后的客户被写入 RHV 导出存储域。还必须使用-os 参数来指定导出存储域的位置。请注意，这实际上并没有将来宾导入 RHV。您必须稍后使用 UI 手动完成该操作。一般使用这个选项将虚拟机转换并进行 Ovirt 的导入。
- vdsm：将输出方法设置为 vdsm。 此模式类似于-o rhv，但必须给出 data

domain 的完整路径:/rhv/data-center/< data-center-uuid >/< data-domain-uuid >。该模式仅在 virt-v2v 在 VDSM 控制下运行时使用。

VMware vCenter 转换到 libvirt 样例

场景样例

VMware vCenter server 名称：vcenter.example.com

数据中心名称：Datacenter

虚拟机管理程序名称：esxi

虚拟机名称：vmware_guest

命令样例

```
virt-v2v -ic vpx://vcenter.example.com/Datacenter/esxi vmware_guest
```

默认输出选项为-o libvirt，默认将虚拟机转换为在本地 libvirt 下运行。

本地磁盘映像转换样例

场景样例

本地磁盘映像路径：/tmp/disk.img

转换后保存路径：/var/tmp/

命令样例

```
virt-v2v -i disk disk.img -o local -os /var/tmp
```

4.10.2. 热迁移增强

当虚拟机在物理机上运行时，物理机可能存在资源分配不均，造成负载过重或过轻的情况。另外，物理机存在硬件更换、软件升级、组网调整、故障处理等操作，如何在不中断业务的情况下完成这些操作十分重要。虚拟机热迁移技术可以在业务连续前提下，完成负载均衡或上述操作，提升用户体验和工作效率。虚拟机热迁移通常是将整个虚拟机的运行状态完整保存下来，同时可以快速恢复到原有的甚至不同的硬件平台上。虚拟机恢复后，仍然可以平滑运行，用户感知不到任何差异。根据虚拟机数据存储在当前主机还是远端存储设备（共享存储）的不同。

热迁移脏页率预查询

用户在迁移前可以使用 dirtyrate 功能，获取热迁移的内存脏页变化速率，根据虚拟机内存使用情况评估虚拟机是否适合迁移或配置更合理的迁移参数

使用方法

例如，指定名为 test 的虚拟机，计算时间为 1s：

```
    #virsh qemu-monitor-command test
'{"execute":"calc-dirty-rate","arguments":{"calc-tim
e":1}}'
```

间隔 1s 后，查询脏页变化速率

```
    #virsh qemu-monitor-command test
'{"execute":"query-dirty-rate"}'
```

vCPU 降频

用户在迁移前可以将 auto-converge 属性开启，减小虚拟机 vcpu 的执行时间，进而达到减小脏页产生的目的。

使用方法

```
    # virsh qemu-monitor-command test --hmp
migrate_set_capability auto-converge on
```

设置 auto-converge 打开状态，可以看到 auto-converge 包含两个可调参量：

cpu-throttle-initial：初始降低 vcpu 性能的百分比，默认值是 20%

cpu-throttle-increment：如果迁移不成功，每次增加对 vcpu 性能的限制百分比，默认值是 10%。

```
    # virsh qemu-monitor-command test --hmp
migrate_set_parameter cpu-throttle-initial 20
    # virsh qemu-monitor-command test --hmp
migrate_set_parameter cpu-throttle-increment 10
```

LTS 多线程

使用方法

```
    #virsh qemu-monitor-command test-migrate
--hmp migrate_set_capability compress on
```

4.10.3. DirtyLimit 的虚拟机热迁移加速

DirtyLimit 热迁移技术是 qemu 中最新的一种迁移节流策略，迁移节流即在虚拟机热迁移过程中减少 vcpu 的运行时间,从而达到降低产生新的内存脏页的速度的目地。之前的迁移节流策略是 auto-converge 方法，由于对虚机内部内存脏页生成速度感知能力不足，节流可能不及时，导致迁移时长过长甚至迁移失败，另外，在在节流时对所有 vcpu 都无差别的减少运行时间，大大减弱迁移时虚机的性能。DirtyLimit 通过 Intel 平台宿主机操作系统内核中的 dirty-ring 特性,能够获得虚拟机内产生脏页的速度大小，以单个 vcpu 的粒度精准节流，减少内存脏页产生速度大的虚拟机迁移时间，提升迁移中的虚拟机性能。

使用方法：

安装 libvirt，libvirt-client 软件包，在 livbirtd.service 启动的情况下使用

1. 开启虚拟机前使用 virtsh edit <vm_name>打开<vm_name>对应的 xml 配置文件，找到<feature> xml 元素，添加子元素

```
<kvm>
  <dirty-ring state='on' size='4096'/>
</kvm>
```

size 可用范围为[1024,65535]，大内存虚拟机可适当增大 size

2. 启动虚拟机

3. 迁移前准备，迁移方式可以是共享存储热迁移，也可以是跨存储热迁移，首先用户通过 qemu 的 qmp 命令查询当前虚拟机的脏页产生速度，virsh 发送 qmp 命令。

```
#先计算,计算时间为 1s，计算方式为 dirty-ring 方式
virsh qemu-monitor-command <vm_name> --pretty '{"execute":
"calc-dirty-rate", "arguments": {"calc-time": 1,"mode": "dirty-ring"}}'
#查询每个 vcpu 脏页产生速度和所有脏页产生速度之和,单位为 MB/s
virsh qemu-monitor-command <vm_name> --pretty '{ "execute":
"query-dirty-rate" }'
```

其次开启 dirty-limit 迁移参数

```
virsh qemu-monitor-command home-vm1 --pretty
'{ "execute":
"migrate-set-capabilities" ,"arguments":{ "capabilities":
[ { "capability": "dirty-limit", "state": true } ] } }'
```

综合考虑发送端到目的端的网络传输速度大小与虚拟机脏页产生速度去设置每个 vcpu 的脏页产生速度上限，速度上限单位为 MB/s。

```
#命令中<value>为上限值，实际使用中注意设置为一个数值，不需要<>符号。
virsh qemu-monitor-command home-vm1 --pretty '{ "execute":
"migrate-set-parameters" ,"arguments": { "vcpu-dirty-limit": <value> } }'
```

除了 auto-converge 迁移参数与 dirty-limit 存在冲突外，其余的迁移参数都可以在迁移时被设置去增强热迁移功能，比如压缩算法参数、多通道迁移参数。

4. 发起迁移

```
#<url>指发送地址，可以是 qemu+tcp 或 qemu+tls 或 qemu+ssh，比如：
qemu+ssh://<target-域名>/system
#--verbose 参数以百分比的形式显示迁移进度
virsh migrate --live  <vm_name>    <url>    --verbose
```

5. 迁移命令完成后显示【100%】

4.10.4. 跨系统热迁移

支持用户将虚拟机从 openEuler 22.03 系列或 CentOS8 系列系统上跨系统热迁移到银河麒麟云底座操作系统 V10 2406 上。

跨系统热迁移的使用方式和通常的热迁移使用方式相同，需要满足以下前提要求：

热迁移两端系统需要配置为相同的 selinux 状态。

银河麒麟云底座操作系统 V10 2406 需要在防火墙软件中放行用于传输热迁移数据的 tcp 端口，或者关闭防火墙服务。

待迁移虚拟机的磁盘应存放与共享存储空间中，共享存储空间需要在银河麒麟云底座操作系统 V10 2406 上挂载在合适的位置，使得两端系统访问虚拟机磁盘的绝对路径相同。

在以下表格的示例场景下，在热迁移源端系统上执行热迁移操作的命令如下，输入热迁移目标系统的 root 用户密码后即会开始热迁移流程。

热迁移目标系统 IP	192.168.1.1
热迁移虚拟机名称	migration-test

热迁移目标系统认证方式	ssh
热迁移数据传输 tcp 端口	49152
是否在目标系统持久保留该虚拟机	是

```
# virsh migrate migration-test qemu+ssh://192.168.1.1/system --live --persistent
--migrateuri tcp://192.168.1.1:49152
```

4.11. 磁盘 IO 悬挂

存储故障（比如存储断链）场景下，物理磁盘的 IO 错误，通过虚拟化层传给虚拟机前端，虚拟机内部收到 IO 错误，可能导致虚拟机内部的用户文件系统变成 read-only 状态，需要重启虚拟机或者用户手动恢复，这给用户带来额外的工作量。

这种情况下，Qemu 提供了一种磁盘 IO 悬挂的能力，即当存储故障时，虚拟机 IO 下发到主机侧时将 IO 悬挂住，在悬挂时间内不对虚拟机内部返回 IO 错误，这样虚拟机内部的文件系统就不会因为 IO 错误而变为只读状态，而是呈现为 Hang 住；同时虚拟机后端按指定的悬挂间隔对 IO 进行重试。如果存储故障在悬挂时间内恢复正常，悬挂住的 IO 即可恢复落盘，虚拟机内部文件系统自动恢复运行，不需要重启虚拟机；如果存储故障在悬挂时间内未能恢复正常，则上报错误给虚拟机内部，通知给用户。

使用方法：

1. 使用 qemu 命令行进行配置。

```
    -drive
file=/path/to/your/storage,format=raw,if=none,id=drive-virti
o-disk0,cache=none,aio=native -device
virtio-blk-pci,scsi=off,bus=pci.0,addr=0x6,
drive=drive-virtio-disk0,id=virtio-disk0,write-cache=on, \
    werror=retry,rerror=retry,retry_interval=2000,retry_time
out=10000
```

2. 使用 xml 文件配置，使用 virsh 命令启动虚拟机

```
    <disk type='block' device='disk'>
     <driver name='qemu' type='raw' cache='none'
io='native' error_policy='retry' rerror_policy='retry'
retry_interval='2000' retry_timeout='10000'/>
     <source dev='/path/to/your/storage'/>
     <target dev='vdb' bus='virtio'/>
     <backingStore/>
    </disk>
```

3. 参数含义：

retry_interval：为 IO 错误重试的间隔，单位：毫秒

retry_time：为 IO 错误重试超时时间，0 表示不会超时，未配置时默认为 0

4.12. VFIO 设备透传

VFIO（Virtual Function I/O）驱动框架是一个用户态驱动框架，在 intel 平台它充分利用了 VT-d 等技术提供的 DMA Remapping 和 Interrupt Remapping 特性，在 arm 平台下这些功能则由 SMMU 提供， 在保证直通设备的 DMA 安全性同时可以达到接近物理设备的 I/O 的性能。VFIO 是一个可以安全的把设备 I/O、中断、DMA 等暴露到用户空间，用户态进程可以直接使用 VFIO 驱动访问硬件，从而可以在用户空间完成设备驱动的框架。

通过 vfio 的方式透传 pci 设备给虚拟机可以较大的提高 pci 设备的性能，直通的步骤如下，

1. 检查是否支持 vfio 驱动

执行命令

```
    # readlink
/sys/bus/pci/devices/0000\:21\:00.0/iommu_group
    ../../../../kernel/iommu_groups/50
```

其中 0000\:21\:00.0 为 pci 设备的 BDF 号，可以通过 lspci 查到。

如果找没有输出信息则代表不支持 iommu 或 smmu 功能

a. 在 x86 平台开启 iommu

首先确认 bios 开启 iommu 支持，然后再添加 grub 参数，在 amd 的 cpu 上运行

的系统/etc/default/grub 文件中添加

```
amd_iommu=on iommu=pt
```

添加完后需要生成对应的配置文件并重启方可生效，生效后执行命令载入模块。

```
modprobe vfio_iommu_type1
modprobe vfio_pci
```

在 intel 的 cpu 运行的系统/etc/default/grub 文件中添加

```
intel_iommu=on iommu=pt
```

生效方式与载入模块方式同上。

b. 在 arm 开启 smmu 支持。

首先确认 bios 开启 iommu 支持，然后再添加 grub 参数，在 amd 的 cpu 上运行的系统/etc/default/grub 文件中添加。

```
iommu.passthrough=1
```

生效方式与载入模块方式同上。

2. 创建 xml 文件，其中 domain，bus，slot，function 均来自被透传的 pci 设备。

```
<hostdev mode='subsystem' type='pci' managed='yes'>
 <driver name='vfio'/>
 <source>
  <address domain='0x0000'
    bus='0x06'
    slot='0x12'
    function='0x5'/>
 </source>
</hostdev>
```

3. 添加到虚拟机配置中。

```
virsh attach-device domain_name domain_name.xml --config
```

4. 重启虚拟机后生效。

4.13. VMTOP 监测工具

vmtop 是运行在宿主机 host 上的用户态工具，通过命令行的方式监控虚拟机占用宿主机的资源情况。例如 CPU 占用率、内存占用率、vCPU 陷入陷出次数等。因此，可以使用 vmtop 作为虚拟化问题定位和性能调优的工具。使用方式如下：

```
vmtop [选项]
```

选项说明

- -d: 设置显示刷新的时间间隔，单位：秒
- -H: 显示虚拟机的线程信息
- -n: 设置显示刷新的次数，刷新完成后退出
- -b: Batch 模式显示，可以用于重定向到文件
- -h: 显示帮助信息
- -v: 显示版本

具体监控项如下:

- VM/task-name: 虚拟机/进程名称
- DID: 虚拟机 id
- PID: 虚拟机 qemu 进程的 pid
- %CPU: 进程的 CPU 占用率
- EXThvc: hvc-exit 次数（采样差）
- EXTwfe: wfe-exit 次数（采样差）
- EXTwfi: wfi-exit 次数（采样差）
- EXTmmioU: mmioU-exit 次数（采样差）
- EXTmmioK: mmioK-exit 次数（采样差）
- EXTfp: fp-exit 次数（采样差）
- EXTirq: irq-exit 次数（采样差）
- EXTsys64: sys64 exit 次数（采样差）
- EXTmabt: mem abort exit 次数（采样差）
- EXTsum: kvm exit 总次数（采样差）
- S: 进程状态
- P: 进程所占用的物理 CPU
- %ST: 被抢占时间与 cpu 运行时间的比，kvm 数据

- %GUE: 虚拟机内部占用时间与 CPU 运行时间的比，kvm 数据
- %HYP: 虚拟化开销占比，kvm 数据

监控项在不同架构会有所不同，支持 arm 与 x86。

在 vmtop 运行状态下使用的快捷键:

- H: 显示或关闭虚拟机线程信息，默认显示该信息
- up/down: 向上/向下移动显示的虚拟机列表
- left/right: 向左/向右移动显示的信息，从而显示因屏幕宽度被隐藏的列
- f: 进入监控项编辑模式，选择要开启的监控项
- q: 退出 vmtop 进程

5. 容器运行时

这个章节主要介绍容器运行时的安装和配置，容器运行时遵循 OCI（Open Container Initial）组成提出的运行时规范和镜像规范，具有控制容器运行的整个生命周期的作用。

5.1. 使用 docker-ce

5.1.1. 简介

docker 是一个开源的应用容器引擎，基于 go 语言开发并遵循了 apache2.0 协议开源。docker 是一种容器技术，它可以对软件及其依赖进行标准化的打包；容器之间相互独立，基于容器运行的应用之间也是相互隔离的；并且容器之间是共享一个 OS kernel 的，充分利用服务器资源，容器可以运行在很多主流的操作系统之上。docker-ce 是 docker 经 docker 团队认证和维护的社区版发行包。

5.1.2. 安装及配置

使用 dnf 工具安装软件包

```
dnf install docker-ce-*  runc -y
```

5.1.3. docker 命令简介

docker commands 命令格式如下：

```
docker [选项] 子命令 [子选项] ...
```

docker commands 命令选项如下表所示。

子命令	功能说明
attach	将本地标准输入输出和错误流附到一个运行中的容器
build	通过 dockerfile 构建镜像
commit	基于一个已修改的容器创建一个镜像
cp	在容器和本地文件系统间拷贝文件或目录
create	创建一个容器
diff	检查容器的文件系统
events	从 docker 服务获取实时时间
exec	在运行容器中运行命令
export	以 tar 格式导出容器文件系统
history	列出镜像历史
images	列出镜像列表
import	导入镜像
info	列出系统级服务信息
inspect	列出 docker 对象信息
kill	kill 掉一个或多个运行的容器
load	从 tar 包或标准输入加载镜像
login	登录镜像仓库

logs	列出容器日志
pause	停止一个或多个容器中的所有进程
port	列出端口映射或容器的特定映射
ps	列出容器
pull	拉取镜像
push	推送镜像
rename	为容器重命名
restart	重启容器
rm	删除容器
rmi	删除镜像
run	在新容器中运行命令
save	将一个或多个镜像保存到 tar 存档中
search	从镜像仓库中搜索镜像
start	启动一个或多个容器
stats	列出容器资源使用统计数据的实时流
stop	停止一个或多个容器
tag	创建一个引用 SOURCE_IMAGE 的标记 TARGET_IMAGE
top	显示容器的运行进程
unpause	取消暂停一个或多个容器中的所有进程
update	更新容器配置
version	列出 docker 版本信息
wait	阻塞直到一个或多个容器停止，打印退出代码

5.1.4. dockerd 配置方法

1）配置存储驱动

```
cat /etc/docker/daemon.json
{
   "storage-driver": "overlay2"
}
```

2）配置默认网桥 ip

```
cat /etc/docker/daemon.json
{
   "bip": "99.5.0.1/16"
}
```

3）配置 systemd 驱动

```
cat /etc/docker/daemon.json
{
   "exec-opts": ["native.cgroupdriver=systemd"]
}
```

4）配置私有镜像仓库

```
cat /etc/docker/daemon.json
{
    "registry-mirrors": ["https://alzgoonw.mirror.aliyuncs.com"]
}
```

5.1.5. 守护进程

启动：

```
systemctl start docker
```

开机自动启动：

```
systemctl enable docker
```

停止：

```
systemctl stop docker
```

重启：

```
systemctl restart docker
```

5.2. 使用 containerd

5.2.1. 简介

containerd 是一个行业标准的容器运行时，强调简单、健壮和可移植性。它可以作为 Linux 和 Windows 的守护进程使用，可以管理其主机系统的完整容器生命周期：映像传输和存储、容器执行和监督、低级存储和网络附件等。

5.2.2. 安装及配置

使用 dnf 工具安装软件包

```
dnf install containerd  runc -y
```

5.2.3. ctr 命令简介

ctr commands 命令格式如下：

```
ctr [选项] 子命令 [子选项] ...
```

ctr commands 命令选项如下表所示。

命令	说明	操作示例
version	打印客户端和服务端版本	ctr version
containers,c,container	管理容器	ctr container list
content	管理内容	ctr content list
events,event	打印 containerd 容器事件	ctr events
images,image,i	管理镜像	ctr image list
leases	管理租赁	ctr leases list
namespaces,namespace,ns	管理命名空间	ctr namespaces list
run	运行容器	ctr run -d image name
snapshots,snapshot	管理快照	ctr snapshot ls
tasks,t,task	管理任务	ctr task ls
plugins,plugin	提供了有关 containerd 插件的信息	ctr plugins ls
pprof	为 containerd 提供 golang-prof 输出	
install	安装新程序包	
oci	OCI 工具	
shim	直接与 shim 交互	
help,h	显示命令列表或一个命令的帮助	ctr --help ctr commands --help

5.2.4. containerd 配置方法

containerd 的配置文件位置：/etc/containerd/config.toml

toml 文件用于指定配置 containerd 的选项，供守护进程使用，若文件不存在或是没有通过--config 选项，将会动态生成默认配置设置。

containerd 提供列出默认配置的功能，可通过如下命令获取默认配置回显：

```
containerd config default
```

用于配置 containerd 守护进程的 TOML 文件有一个简短的全局设置列表，后面是针对特定区域的一系列守护进程配置。TOML 文件中包含关于**plugin**的部分，允许每个 containerd 插件都有一个独立的插件配置设定区域。

（1）version：配置信息版本，如果未指定则为 1，但需注意 1 版本已被弃用

（2）root：containerd 元数据的根目录。（默认/var/lib/containerd）

（3）plugin_dir：动态插件存放目录，用于存放如 grpc、ttrpc、debug 等插件

5.2.5. 守护进程

启动：

```
systemctl start containerd
```

自动启动：

```
systemctl enable containerd
```

停止：

```
systemctl stop containerd
```

重启：

```
systemctl restart containerd
```

5.3. 使用 podman

5.3.1. 简介

Podman 是一个用于管理容器和映像、挂载到这些容器中的卷以及由容器组组成的

POD 的工具。Podman 基于 libpod, libpod 是一个用于容器生命周期管理的库，也包含在这个存储库中。libpod 库提供了用于管理容器、pod、容器映像和卷的 api。

5.3.2. 安装及配置

使用 dnf 工具安装软件包

```
dnf install podman -y
```

5.3.3. podman 命令简介

podman commands 命令格式如下：

```
podman [选项] 子命令 [子选项] ...
```

podman commands 命令选项如下表所示。

命令	说明	操作示例
attach	将本地标准输入输出和错误流附到一个运行中的容器	podman attach ctrID
auto-update	根据容器的自动更新策略自动更新容器	podman auto-update
build	使用 containerfiles 构建镜像	podman build -f Containerfile.simple .
commit	基于一个已修改的容器创建一个镜像	podman commit containerID
container	容器管理	podman container list
cp	在容器和本地文件间拷贝文件或目录	
create	创建容器	podman create alpine ls
diff	检查容器或镜像文件系统改变	podman diff imageID
events	监控 podman 事件	podman events
exec	在容器中运行命令	podman exec -it ctrID ls
export	以 tar 格式导出容器文件系统	podman export ctrID > myCtr.tar
generate	生成 container、pod 或 volume 等对象的结构化数据	podman generate kube podID
healthcheck	运行容器健康检查	podman healthcheck run mycontainer
history	列出镜像历史	podman history docker.io/library/busybo

		x
image	镜像管理	podman image list
images	列出镜像	podman images --format json
import	导入镜像	cat ctr.tar | podman import -
info	列出系统级信息	
init	创建一个或多个容器 OCI spec	podman init mycontainer
inspect	列出 podman 相关对象信息	podman inspect myvolume
kill	kill 一个或多个容器	podman kill mycontainer
load	从 tar 包加载镜像	podman load -i ctr.tar
login	登录镜像仓库	
logout	登出镜像仓库	
logs	列出容器日志	podman logs ctrID
machine	虚拟机管理	podman machine init myvm
manifest	manifest 管理	podman mainifest rm mylist:v1
mount	mount 容器的根文件系统	
network	网络管理	podman network ls
pause	停止一个或多个容器中的所有进程	podman pause -a
play	应用 container、pods、volume 结构化文件	podman play kube nginx.yml
pod	pod 管理	podman pod ps
port	列出或指定端口映射或容器的特定映射	podman port --all
ps	列出容器	
pull	拉取镜像	
push	推送镜像	
rename	重命名一个存在镜像	
restart	重启容器	
rm	删除容器	
rmi	删除镜像	
run	在容器中运行命令	podman run imageID ls -alF /etc
save	将一个或多个镜像保存到 tar 存档中	podman save > alpine-all.tar alpine:latest
search	从镜像仓库中搜索镜像	
secret	秘钥管理	podman secret ls
start	启动容器	podman start --interactive --attach imageID
stats	列出容器资源使用统计数据的实时流	podman stats ctrID

stop	停止容器	podman stop ctrID
system	podman 管理	podman system df
tag	为本地镜像添加别名	podman tag imageID:latest myNewImage:newTag
top	列出容器运行的进程	podman top ctrID -eo user,pid,comm
unmount	清除容器文件系统挂载点	podman unmount ctrID1 ctrID2 ctrID3
unshare	在修改后的用户命名空间中运行命令	podman unshare ctrID
unpause	取消暂停一个或多个容器中的所有进程	podman unpause ctrID
untag	清除镜像别名	podman untag imageID:latest otherImageName:latest
version	列出 podman 版本信息	
volume	volume 管理	podman volume ls
wait	阻塞直到一个或多个容器停止，打印退出代码	podman wait --interval 5s ctrID

5.3.4. 用户配置

有根环境中 podman 配置文件位于/usr/share/containers 和/etc/containers 中，无根环境配置文件通常位于~/.config/containers（$[XDG_CONFIG_HOME]/containers）。

（1）container.conf 用于设置 container 相关配置，位置：

/usr/share/containers/containers.conf

/etc/containers/containers.conf

$HOME/.config/containers/containers.conf

（2）storage.conf 用于设置存储相关配置，位置：

/etc/containers/storage.conf

$HOME/.config/containers/storage.conf

（3）registries.conf 用于设置镜像仓库相关配置，位置：

/etc/containers/registries.conf

/etc/containers/registries.d

$HOME/.config/containers/registries.conf

5.4. 使用 kata-container

5.4.1. 配置文件

(1) 安装 kata-containers 相关包

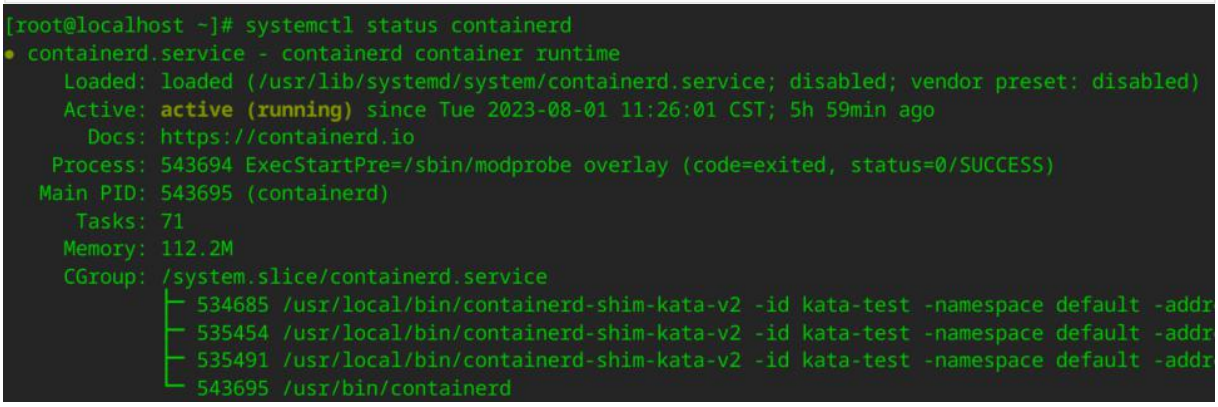
```
$ yum install kata-containers-3.0.0-2.ky10h.x86_64.rpm
$ yum install containerd docker-runc
$ yum install qemu
```

(2) 修改 containerd 的 config 文件,设置 kata 容器运行时

```
$ cat /etc/containerd/config.toml
 [plugins]
   [plugins."io.containerd.grpc.v1.cri"]
     [plugins."io.containerd.grpc.v1.cri".containerd]
       default_runtime_name = "kata"
       [plugins."io.containerd.grpc.v1.cri".containerd.runtimes]
[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.kata]
           runtime_type = "io.containerd.kata.v2
```

(3) 重启 containerd 使配置生效

```
$ systemctl daemon-reload && systemctl restart containerd
```



5.4.2. 功能使用

(1) 以 busybox 镜像为例：

```
$ ctr images pull docker.io/library/busybox:latest
```

(2) 启动一个 kata 安全容器，查看 linux 版本

```
$ ctr run --runtime=io.containerd.kata.v2 --rm
docker.io/library/busybox:latest kata-test uname -a
```

```
[root@localhost kata-containers]# uname -a
Linux localhost 5.10.0-153.5.v2305.ky10h.x86_64 #1 SMP Fri Jun 30 14:22:44 CST 2023 x86_64 x86_64 x86_64 GNU/Linux
[root@localhost kata-containers]# ctr run --rm --runtime=io.containerd.kata.v2 docker.io/library/busybox:latest kata-test-1 uname -a
Linux localhost 5.10.134 #4 SMP Mon Oct 17 17:53:10 CST 2022 x86_64 GNU/Linux
```

(3) 以 nginx 镜像为例，验证网络功能

```
$ ctrimagepulldocker.io/library/nginx:latest
```

(4) 后台运行 nginx 测试容器

```
$ ctr run -d --runtime=io.containerd.kata.v2
docker.io/library/nginx:latest nginx-test
```

```
[root@localhost hym]# ctr run -d --runtime=io.containerd.kata.v2 docker.io/library/nginx:latest nginx-test
[root@localhost hym]# ctr c ls
CONTAINER     IMAGE                                RUNTIME
kata-test     docker.io/library/busybox:latest     io.containerd.kata.v2
nginx-test    docker.io/library/nginx:latest       io.containerd.kata.v2
[root@localhost hym]# ctr task ls
TASK          PID       STATUS
nginx-test    640595    RUNNING
```

(5) 进入 nginx 测试容器

```
$ ctr task  exec -t --exec-id 0 nginx-test sh
```

(6) 查看网络状态

```
$ curl 127.0.0.1/index.html
```

```
# curl 127.0.0.1/index.html

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

(7) virtiofs 使 guest 和宿主机之间共享文件系统，查看文件系统状态

```
1 # 宿主机
2 $ df-h
```

```
[root@localhost kata-containers]# df -h
文件系统             容量  已用  可用 已用% 挂载点
devtmpfs             4.0M     0  4.0M    0% /dev
tmpfs                126G  137M  126G    1% /dev/shm
tmpfs                 51G  147M   50G    1% /run
tmpfs                4.0M     0  4.0M    0% /sys/fs/cgroup
/dev/mapper/kh-root  5.7T  329G  5.4T    6% /
tmpfs                126G   16K  126G    1% /tmp
/dev/sda2           1006M  214M  793M   22% /boot
tmpfs                 26G  8.0K   26G    1% /run/user/0
overlay              5.7T  329G  5.4T    6% /run/containerd/io.containerd.runtime.v2.task/default/nginx-test/rootfs
```

```
1 # 容器
2 $ df-h
```

```
[root@localhost kata-containers]# ctr task  exec -t --exec-id 0 nginx-test sh
# df -h
Filesystem      Size  Used Avail Use% Mounted on
none            5.7T  328G  5.4T   6% /
tmpfs            64M     0   64M   0% /dev
shm             985M     0  985M   0% /dev/shm
tmpfs            64M  4.0K   64M   1% /run
```

6. 云组件

6.1. kubernetes 部署使用

6.1.1. 部署 (all-in-one)

基础配置

1. 修改主机名

```
hostnamectl set-hostname --static master
```

2. 配置 yum 源

3. 关闭防火墙

```
systemctl disable firewalld
systemctl stop firewalld
```

4. 关闭 swap

```
swapoff -a
```

5. 修改/etc/hosts

将 ip 与主机名的映射关系写入到/etc/hosts 文件里，配置 IPv4 转发并让 iptable 看到桥接流量

```
cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
overlay
br_netfilter
EOF

sudo modprobe overlay
sudo modprobe br_netfilter
```

设置所需 sysctl 参数

```
vim /etc/sysctl.conf
net.bridge.bridge-nf-call-iptables  = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward                 = 1

sysctl -p
```

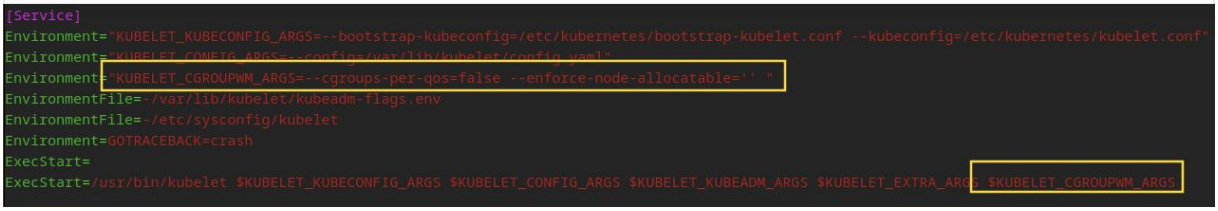
安装 kubernetes-1.26.7 软件包

yum install kubernetes-master-1.26.7-1.ky10h.x86_64.rpm
kubernetes-node-1.26.7-1.ky10h.x86_64.rpm
kubernetes-client-1.26.7-1.ky10h.x86_64.rpm
kubernetes-kubeadm-1.26.7-1.ky10h.x86_64.rpm

关闭 kubelet 对工作负载下 pod 的 cgroupfs 管理

禁用 cgroups-per-qos

vim /etc/systemd/system/kubelet.service.d/kubeadm.conf



安装容器运行时 crio 软件包

```
yum install cri-o conmon runc
systemctl start crio
```

初始化 kubeadm 引导文件

```
# kubeadm config print init-defaults > kubeadm-init.yaml
修改 advertiseAddress: 字段，指定本地网卡地址
修改 criSocket: 字段，指定 crio.sock 文件描述符
修改 name: 字段，指定本机 hostname
networking 中增加 podSubnet，指定与网络插件同子网
例：
advertiseAddress: 192.168.122.121
kubernetesVersion: v1.26.7
podSubnet: 10.244.0.0/16
serviceSubnet: 10.96.0.0/12
```

获取部署 kubernetes 必需的容器镜像

查看所需容器镜像

```
kubeadm config images list --config kubeadm-init.yaml
```

获取所需容器镜像

```
kubeadm config images pull --config kubeadm-init.yaml
```

配置 registry 地址（可选）

修改 crio pause_image 版本

```
# vim /etc/crio/crio.conf
pause_image = "10.41.160.159:4000/pause:3.9"
```

部署 kubernetes all-in-one

```
# 初始化集群 controller 节点
kubeadm init --config kubeadm-init.yaml
# 配置 cluster 访问
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

```
Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

Alternatively, if you are the root user, you can run:

  export KUBECONFIG=/etc/kubernetes/admin.conf

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 10.41.160.211:6443 --token abcdef.0123456789abcdef \
        --discovery-token-ca-cert-hash sha256:519aa31cee5ca62706468eada499c760f1d7cc641bb43a51f29c8cbdf602d56b
[root@hostos-test deploy-k8s-1.26]#
```

6.1.2. cgroup v2 使用

kubernetes1.26 默认支持 cgroup v2，开启 cgroup v2 方法如下：

修改内核参数：

```
    /etc/default/grub 下的 GRUB_CMDLINE_LINUX 中添加
systemd.unified_cgroup_hierarchy=1
```

重新生成 grub：

```
    grub2-mkconfig -o /boot/grub2/grub.cfg
    reboot
```

设置 kubelet 和容器运行时 cgroup_driver 为 systemd

k8s 使用 cgroup v2 组件版本要求以及配置参考：

https://kubernetes.io/docs/concepts/architecture/cgroups/

6.1.3. crio 功能使用

crictl command 使用方法：

ps	列出所有容器，包括，正在运行的和已经停止的
run	运行一个容器
logs	查看容器的日志
stop	停止容器
rm	删除容器
inspect	查看容器的具体信息，包括（状态，镜像，容器 ID）
images	列出所有容器镜像
attach	附加到某个容器
create	创建一个容器
exec	在容器内执行命令
version	返回容器运行时版本
insepecti	返回镜像的信息
pull	从仓库拉取镜像
runp	启动一个 pod
rmi	删除镜像
rmp	删除 pod

pods	列出所有的 pod
start	启动一个创建好的容器
info	返回容器运行时的信息
stop	停止正在运行的容器
stopp	停止正在运行的 pod
update	升级正在运行的容器
config	获取或者设置 crictl 客户端选项
stats	显示容器的实时性能统计信息
help	获取命令的帮助信息

6.1.4. k8s 管理 pod

k8s 使用 kubectl 命令行工具管理 pod

apply	创建 pod
get	获取 pod 列表
describe	查看 pod 详细信息
delete	删除 pod
exec	在容器内执行命令
create	创建一个 pod（pod.yaml 配置必须是完整的）
scale	扩容或缩容 Deployment、ReplicaSet、Replication Controller 或 Job 中 Pod 数量
rollout	用于管理滚动升级
attach	用于连接到容器的标准输入，标准输出和错误流
lable	用于给资源对象添加或修改标签
annotate	更新一个或多个资源的 Annotations 信息

6.2. kubeedge 部署使用

6.2.1. 环境配置

以下配置在 cloud 和 edge 端都需要配置

（1）关闭防火墙

```
$ systemctl stop firewalld
$ systemctl disable firewalld
```

（2）禁用 selinux，swap

```
$ setenforce 0
$ swapoff -a
```

（3）网络配置，开启相应的转发机制，生效规则

```
$ cat >> /etc/sysctl.d/k8s.conf <<EOF
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
vm.swappiness=0
EOF

$ modprobe br_netfilter
$ sysctl -p /etc/sysctl.d/k8s.conf

$ cat /proc/sys/net/bridge/bridge-nf-call-ip6tables
1
$ cat /proc/sys/net/bridge/bridge-nf-call-iptables
1
```

（4）设置 hostname

```
# 云侧
$ hostnamectl set-hostname cloud
# 边缘侧
$ hostnamectl set-hostname edge
```

（5）配置 hosts 文件（示例），按照用户实际情况设置

```
$ cat >> /etc/hosts << EOF
172.17.66.182 cloud
172.17.66.183 edge
EOF
```

（6）安装 cri-tools 网络工具

```
$ yum install cri-tools -y
$ yum install containernetworking-plugins -y
```

6.2.2. 云侧部署 kubernetes

kubernetes 初始化请参考 7.1 kubernetes 部署使用

kubeedge 支持 flannel 网络插件，拉取 flannel 官方 yaml 文件，因为云侧和边缘侧为不同的网络环境，需要配置不同的亲和性，所以这里需要两份 flannel 配置文件。

（1）部署云侧网络需要修改 yaml 文件，具体修改如下：

修改 metadata-name 来区分 cloud 和 edge 节点，修改 affinity 亲和性，增加一个 key 使其不部署于 edge 节点

```
...
      kind: DaemonSet
      metadata:
     -  name: kube-flannel-ds
     +  name: kube-flannel-cloud-ds
        namespace: kube-system
        labels:
          tier: node
                    operator: In
...
                    values:
                    - linux
     +              - key: node-role.kubernetes.io/agent
     +                operator: DoesNotExist
          hostNetwork: true

     $ cp kube-flannel.yml kube-flannel-cloud.yml
```

（2）部署云侧网络需要修改 yaml 文件，具体修改如下：

修改 metadata-name 区分云和边缘节点，修改 affinity 亲和性，增加一个 key 使其部署于 edge 节点，参数新增--kube-api-url，指定边缘侧 edgecore 监听地址。

```
...
 metadata:
-  name: kube-flannel-ds
+  name: kube-flannel-edge-ds
   namespace: kube-system
   labels:
     tier: node
...
                operator: In
                values:
                - linux
+               - key: node-role.kubernetes.io/agent
+                 operator: Exists
       hostNetwork: true
...
         args:
         - --ip-masq
         - --kube-subnet-mgr
+        - --kube-api-url=http://127.0.0.1:10550

$ cp kube-flannel.yml kube-flannel-edge.yml
```

（3）配置 flannel 网络插件，查看节点状态

```
$ kubectl apply -f kube-flannel-cloud.yml
$ kubectl apply -f kube-flannel-edge.yml
$ kubectl get node -A
```

```
[root@cloud ~]# kubectl get node
NAME              STATUS   ROLES                  AGE    VERSION
cloud.kubeedge    Ready    control-plane,master   4d2h   v1.23.12
```

6.2.3. 云侧部署 cloudcore

(1) 安装软件包

```
$ yum install kubeedge-keadm kubeedge-cloudcore
$ cp /etc/kubeedge/config/cloudcore.example.yaml
/etc/kubeedge/config/cloudcore.yaml
```

(2)修改 cloud kubeedge 的配置文件

kubeConfig 指定云节点 kubernetes 的 kubeconfig 文件地址，advertiseAddress 指定云节点 IP 地址，开启 cloud stream 服务，支持 kubectl logs/exec 功能。

```
$ kubeConfig: "/root/.kube/config"
$     advertiseAddress:
      - <cloudIP>
$   cloudStream:
    enable: true
    streamPort: 10003
```

(3) 生成证书

```
$ mkdir -p /etc/kubeedge/{config,ca,certs}
$ cp /etc/kubeedge/tools/certgen.sh /etc/kubeedge/

# 生成 CA 证书
$ ./certgen.sh genCA

# 证书请求
$ ./certgen.sh genCsr server

# 生成证书
$ ./certgen.sh genCert server <cloudIP>

# stream 证书
$ export CLOUDCOREIPS=<cloudIP>
$ ./certgen.sh stream
```

(4) 启动 cloudcore 服务

```
$ systemctl daemon-reload
$ systemctl restart cloudcore

# 查看 cloudcore 状态
$ systemctl status cloudcore
```

```
[root@cloud config]# systemctl status cloudcore
● cloudcore.service
     Loaded: loaded (/usr/lib/systemd/system/cloudcore.service; enabled; vendor preset: disabled)
     Active: active (running) since Tue 2023-09-12 14:15:39 CST; 3h 3min ago
   Main PID: 3364 (cloudcore)
      Tasks: 14 (limit: 41933)
     Memory: 65.8M
        CPU: 42.124s
     CGroup: /system.slice/cloudcore.service
             └─ 3364 /usr/local/bin/cloudcore
```

(5) 查看端口状态

```
$ ss -nutlp |egrep "10003|10004"
```

```
[root@cloud config]# ss -nutlp |egrep "10003|10004"
tcp   LISTEN 0      4096               *:10003            *:*    users:(("cloudcore",pid=3364,fd=9))
tcp   LISTEN 0      4096               *:10004            *:*    users:(("cloudcore",pid=3364,fd=8))
```

6.2.4. 边缘侧部署 edgecore

(1) 获取云侧证书

```
$ keadm gettoken
53460f773b18ccf191f896bc899b4d8551ffc73e78336a9778427da350b465b1.eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE2OTQyNTA1OTh9.RGQgixsfJgLiQXw6Xjpd03IOyRDOiGiAnUprH5ebvKs
# 云侧证书传给边缘侧
$ tar cvf certs.tar /etc/kubeedge/ca /etc/kubeedge/certs && scp certs.tar <edgeIP>:/tmp
```

(2) 边缘侧加载证书

```
$ tar xvf certs.tar -C /
```

(3) 安装 edgecore 软件包，启动 edgecore 服务

```
$ yum install -y kubeedge-edgecore kubeedge-edgesite kubeedge-keadm
$ yum install -y docker-ce docker-runc
$ systemctl start docker && systemctl start containerd
```

(1) 修改 edgecore 的配置文件

修改 edgeHub 的 httpServer 为边缘侧 IP 地址，修改 token 为从云侧获取的 token，修改 websocket 的 server 为云侧 IP 地址，开启 edgeStream 服务，支持 kubectl logs/exec 功能，新增 clusterDNS 参数，开启 metadata server 服务

```
-apiVersion: edgecore.config.kubeedge.io/v1alpha2
+apiVersion: edgecore.config.kubeedge.io/v1alpha1
  edgeHub:
-   httpServer: https://<edgeIP>:10002
+   httpServer: https://<cloudIP>:10002
-   token: ""
+   token:
"53460f773b18ccf191f896bc899b4d8551ffc73e78336a9778427da350b465b1.eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE2OTQzODAxOTh9.4vFrfjtYziwFJmKI3wxS_1I9zuvntSL27cI3b3cwihM"
    websocket:
-     server: <edgeIP>:10000
+     server: <cloudIP>:10000
      writeDeadline: 15
  edgeStream:
-   enable: false
+   enable: true
    handshakeTimeout: 30
    readDeadline: 15
-   server: 127.0.0.1:10004
+   server: <cloudIP>:10004
+     clusterDNS:
+     - <cloudDNS>
    metaServer:
-     enable: false
+     enable: true
```

(5) 查看 edgecore 状态

```
$ systemctl daemon-reload && systemctl start mosquitto
$ systemctl start edgecore
$ systemctl status edgecore
```

```
[root@edge kubeedge]# systemctl status edgecore
● edgecore.service
     Loaded: loaded (/usr/lib/systemd/system/edgecore.service; disabled; vendor preset: disabled)
     Active: active (running) since Mon 2023-09-11 17:09:01 CST; 24h ago
   Main PID: 199002 (edgecore)
      Tasks: 41 (limit: 41933)
     Memory: 117.3M
     CGroup: /system.slice/edgecore.service
             └─ 199002 /usr/local/bin/edgecore
```

(6) 查看云侧节点状态

```
$ kubectl get node
```

```
[root@cloud config]# kubectl get node
NAME             STATUS   ROLES                  AGE    VERSION
cloud.kubeedge   Ready    control-plane,master   4d3h   v1.23.12
edge             Ready    agent,edge             2d7h   v1.24.14-kubeedge-v1.14.2-dirty
```

7. 网络

7.1. ovs+dpdk 部署使用

7.1.1. 概述

OpenvSwitch+dpdk 广泛用于云业务场景下，提供灵活的虚拟网络及强大的转发性能。这部分结合云业务场景，设计典型的 OpenvSwitch 虚拟机组网，并搭建相应的虚拟网络环境，演示 OpenvSwitch 与 dpdk 的基本使用。

OpenvSwitch（以下简称 ovs）以其丰富的功能，作为多层虚拟交换机，已经广泛应用于云环境中。ovs 的主要功能是为物理机上的 VM 提供二层网络接入，和云环境中的其它物理交换机并行工作在 Layer 2。传统 host 中 ovs 工作在内核态，与 guest virtio 的数据传输过程中需要多次内核态和用户态的数据切换，带来性能瓶颈。

dpdk 是 Intel 公司开源的高性能用户态网络数据平面开发工具集，ovs 是云计算内广泛应用的开源虚拟交换机实现，ovs+dpdk 提供了灵活的网络管理和高性能转发的能力。关于 ovs 和 dpdk 的更多信息请访问 dpdk 官网（https://www.dpdk.org/）和 ovs

官网（https://www.openvswitch.org/）。

7.1.2. 环境要求

7.1.2.1. 硬件要求

类别	配置要求
主机	两台主机-Host1 Host2
CPU	Intel/Hygon/FT-2000+/Kunpeng-920
网卡	支持 dpdk，两台主机的网卡直连

注：以下操作如未有特殊说明，两台主机同步进行。

7.1.2.2. 系统要求

1)设置内存大页，开启 IOMMU 或 SMMU 编辑开机启动项。

机器型号	BIOS 参数(etc/default/grub 中的“GRUB_CMDLINE_LINUX”后添加)
Intel(x86)	default_hugepagesz=[size_page] hugepagesz=[size_page] hugepages=[num_pages] iommu.passthrough=1
Hygon(x86)	default_hugepagesz=[size_page] hugepagesz=[size_page] hugepages=[num_pages] amd_iommu=on iommu=pt
FT-2000+(aarch64)	default_hugepagesz=[size_page] hugepagesz=[size_page]hugepages=[num_pages] iommu.passthrough=1
Kunpeng-920(aarch64)	default_hugepagesz=[size_page] hugepagesz=[size_page] hugepages=[num_pages] iommu.passthrough=1
虚拟机（x86）	default_hugepagesz=[size_page] hugepagesz=[size_page] hugepages=[num_pages] iommu=pt

注：[size_page]代表每页大页的大小，[num_pages] 代表大页数量，二者相乘代表总共开启的大页大小。x86[size_page] 可选 2M 与 1G；aarch64 [size_page]可选 512M。[num_pages] 根据硬件实际情况给定。

UEFI 启动方式写配置：

```
#grub2-mkconfig -o /boot/efi/EFI/kylin/grub.cfg
```

BIOS 启动方式写配置：

```
#grub2-mkconfig -o /boot/grub2/grub.cfg
```

2)关闭防火墙和 selinux

```
#systemctl stop firewalld
#systemctl disable firewalld
#setenforce 0
# sed -i 's#SELINUX=enforcing#SELINUX=disabled#g' /etc/selinux/config
```

3)重启生效。

7.1.3. 安装 ovs 和 dpdk 软件包启动服务

1)安装 ovs 和 dpdk

```
#dnf install openvswitch openvswitch-dpdk dpdk
```

2)修改/etc/sysconfig/openvswitch 配置，修改文件用户与用户组

```
#cat /etc/sysconfig/openvswitch
....
OVS_USER_ID="root:root"
```

3)启动服务

```
#systemctl start openvswitch
#systemctl enable openvswitch
```

4)设置 dpdk 启动参数

```
#ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-init=true
```

5)建一个到 dpdk 默认驱动程序路径的软链接

```
#mkdir -p /usr/local/lib64/dpdk

#ln -fs /usr/lib64/dpdk/pmds-22.0 /usr/local/lib64/dpdk/pmds-22.0
```

6)重启 ovs

```
#systemctl restart openvswitch
```

7)将网卡绑定到 dpdk 用户态

a.安装 NIC 网卡驱动模块 vfio-pci

```
#modprobe vfio-pci
#chmod a+x /dev/vfio
#chmod 0666 /dev/vfio/*
```

注：如果系统没有 IOMMU，启用 noiommu 模式，echo Y > /sys/module/vfio/parameters/enable_unsafe_noiommu_mode

a.查看网口 pci 信息，找到需要绑定网口的 PCI 地址

```
#dpdk-devbind -s
```

b.执行绑定

```
#dpdk-devbind --bind=vfio-pci 0000:00:08.0
#dpdk-devbind --bind=vfio-pci 0000:00:09.0
```

注：0000:00:08.0 和 0000:00:09.0 为网卡 pci 值，请根据实际情况填写

c.再次查看，验证是否绑定成功

```
[root@localhost ~]# dpdk-devbind -s

Network devices using DPDK-compatible driver
============================================
0000:00:08.0 '82540EM Gigabit Ethernet Controller 100e' drv=vfio-pci unused=e1000
0000:00:09.0 '82540EM Gigabit Ethernet Controller 100e' drv=vfio-pci unused=e1000

Network devices using kernel driver
===================================
0000:00:02.0 '82540EM Gigabit Ethernet Controller 100e' if=ens2 drv=e1000 unused=vfio-pci *Active*

No 'Baseband' devices detected
==============================
```

7.1.4. 配置 vxlan 组网

验证是以 openstack neutron，使用 ovs plugin 为基础的经典 vxlan 组网方案拓扑图如下：

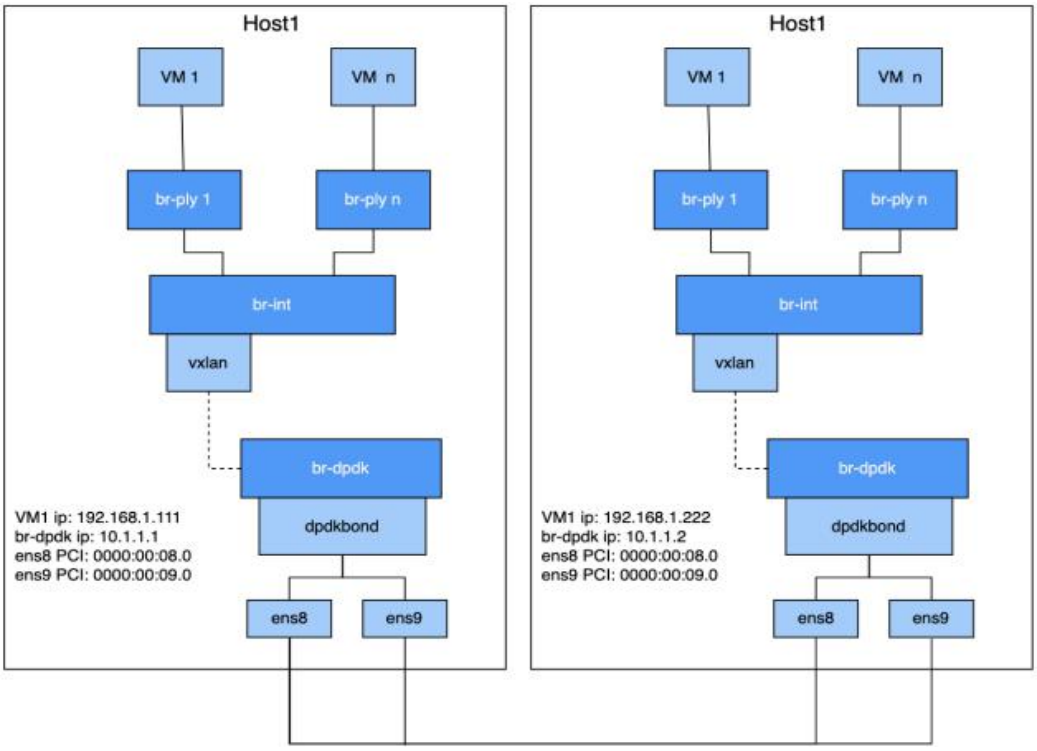


图 7-1 vxlan 组网

以上图为例，图中标出组网配置所需要的参数。以下列出在 Host1 上的操作，在 Host2 上基本一致。

1)添加并配置 br-dpdk 网桥，并配置 bond

```
    #ovs-vsctl add-br br-dpdk -- set bridge br-dpdk datapath_type=netdev
    #ovs-vsctl add-bond br-dpdk dpdk-bond p0 p1 -- set Interface p0
type=dpdk options:dpdk-devargs=0000:00:08.0 -- set Interface p1 type=dpdk
options:dpdk-devargs=0000:00:09.0
    #ovs-vsctl set port dpdk-bond bond_mode=active-backup
    #ovs-vsctl set port dpdk-bond lacp=off
    #ifconfig br-dpdk 10.1.1.1/24 up
```

注：ifconfig br-dpdk 10.1.1.1/24 up 这个是配置 br-dpdk 网桥的 IP 地址，用来配置 VXLAN 隧道。Host2 上执行 ifconfig br-dpdk 10.1.1.2/24 up 来配对，隧道的网段和虚拟机的网段区别开来。dpdk-devargs 传入的 PCI 值请根据真实情况填写。若 dpdk 仅绑定一张网卡，可不进行 bond 配置。

2)添加并配置 br-dpdk 网桥

```
    #ovs-vsctl add-br br-int -- set bridge br-int datapath_type=netdev
    #ovs-vsctl add-port br-int vxlan0 -- set Interface vxlan0 type=vxlan
options:local_ip=10.1.1.1 options:remote_ip=10.1.1.2
```

注：该组网下 br-int 上有一个 VXLAN 端口，出主机的流量都会加上 VXLAN 头。而 VXLAN 口的 local_ip 填的是本主机 br-dpdk 的 IP 地址，remote_ip 是对端 br-dpdk 的 IP 地址。所以 Host1 和 Host2 中 local_ip 和 remote_ip 相反。

3)添加并配置 br-ply1 网桥

```
    #ovs-vsctl add-br br-ply1 -- set bridge br-ply1 datapath_type=netdev
    #ovs-vsctl add-port br-ply1 tap1 -- set Interface tap1
type=dpdkvhostuserclient
options:vhost-server-path=/var/run/openvswitch/tap1
    #ovs-vsctl add-port br-ply1 p-tap1-int -- set Interface p-tap1-int
type=patch options:peer=p-tap1
    #ovs-vsctl add-port br-int p-tap1 -- set Interface p-tap1 type=patch
options:peer=p-tap1-int
```

注：该组网每增加一个虚拟机，就增加一个 br-ply 网桥，该网桥上有一个 dpdkvhostuser 给虚拟机，patch 口连到 br-int 网桥上。

4)验证组网

```
    #ovs-vsctl show
    88515d86-cad2-493d-96e5-f365fbe0b9ec
```

```
    Bridge br-int
        datapath_type: netdev
        Port br-int
            Interface br-int
                type: internal
        Port "vxlan0"
            Interface "vxlan0"
                type: vxlan
                options: {local_ip="10.1.1.1", remote_ip="10.1.1.2"}
        Port "p-tap1"
            Interface "p-tap1"
                type: patch
                options: {peer="p-tap1-int"}
    Bridge "br-ply1"
        datapath_type: netdev
        Port "br-ply1"
            Interface "br-ply1"
                type: internal
        Port "tap1"
            Interface "tap1"
                type: dpdkvhostuserclient
                options: {vhost-server-path="/var/run/openvswitch/tap1"}
        Port "p-tap1-int"
            Interface "p-tap1-int"
                type: patch
                options: {peer="p-tap1"}
    Bridge br-dpdk
        datapath_type: netdev
        Port dpdk-bond
            Interface "p0"
                type: dpdk
                options: {dpdk-devargs="0000:00:08.0"}
            Interface "p1"
                type: dpdk
                options: {dpdk-devargs="0000:00:09.0"}
        Port br-dpdk
            Interface br-dpdk
                type: internal
    ovs_version: "2.12.0"
```

5)验证 Host1 和 Host2 两端的 br-dpdk 网桥是否连通

```
#ping 10.1.1.2
```

```
[root@localhost ~]# ping 10.1.1.2
PING 10.1.1.2 (10.1.1.2) 56(84) bytes of data.
64 bytes from 10.1.1.2: icmp_seq=1 ttl=64 time=3.03 ms
64 bytes from 10.1.1.2: icmp_seq=2 ttl=64 time=0.438 ms
64 bytes from 10.1.1.2: icmp_seq=3 ttl=64 time=0.315 ms
64 bytes from 10.1.1.2: icmp_seq=4 ttl=64 time=0.304 ms
```

6)启动虚拟机

虚拟机配置需要注意内存大页、网口配置，以下是虚拟机配置可参考:

```
    .........
    <memory unit='KiB'>2097152</memory>
    <currentMemory unit='KiB'>2097152</currentMemory>
    <memoryBacking>
        <hugepages>
            <page size='1048576' unit='KiB'/>
        </hugepages>
    </memoryBacking>
    <vcpu placement='static' cpuset='20-21'>2</vcpu>
    <cputune>
        <vcpupin vcpu='0' cpuset='20'/>
        <vcpupin vcpu='1' cpuset='21'/>
    </cputune>
    <numatune>
    <memory mode='strict' nodeset='2'/>
    </numatune>
    .....
    <cpu mode='host-passthrough' check='none'>
        <topology sockets='1' dies='1' cores='2' threads='1'/>
        <numa>
            <cell id='0' cpus='0-1' memory='2097152' unit='KiB'
memAccess='shared'/>
        </numa>
    </cpu>
    ..............
    <interface type='vhostuser'>
        <mac address='52:54:00:6e:51:c9'/>
        <source type='unix' path='/var/run/openvswitch/tap1'
mode='server'/>
        <target dev='tap1'/>
        <model type='virtio'/>
        <driver name='vhost' queues='2' rx_queue_size='1024'
tx_queue_size='1024'/>
    </interface>
```

- memoryBacking 标签指明在主机哪种规格内存大页上申请内存，由于主机配置的 1G 内存大页，所以这里保持一致。

- numatune 标签指明内存在主机哪个 node 上申请，这里和网卡 node 保持一致。

- numa 子标签指明虚拟机内存的模式，这里配置了 vhostuser 的虚拟网口，虚拟机需要有共享的内存大页。

- interface 标签定义虚拟机的虚拟网口：

  √ source 的 path 指定了主机和虚拟机通信的套接字文件位置，mode 指明虚拟机套接字类型，由于 ovs 端配置的是 dpdkvhostuserclient，为客户端模式，所以这里配置服务模式。

√ target 指明虚拟机使用的套接字名称。

√ driver 指明虚拟机使用的驱动，并指明了队列及队列深度。这里使用的是 vhostuser 的驱动。

√ qemu 进程需要开启 root 权限。

1)跨主机虚拟机连通性验证

进入虚拟机关闭防火墙

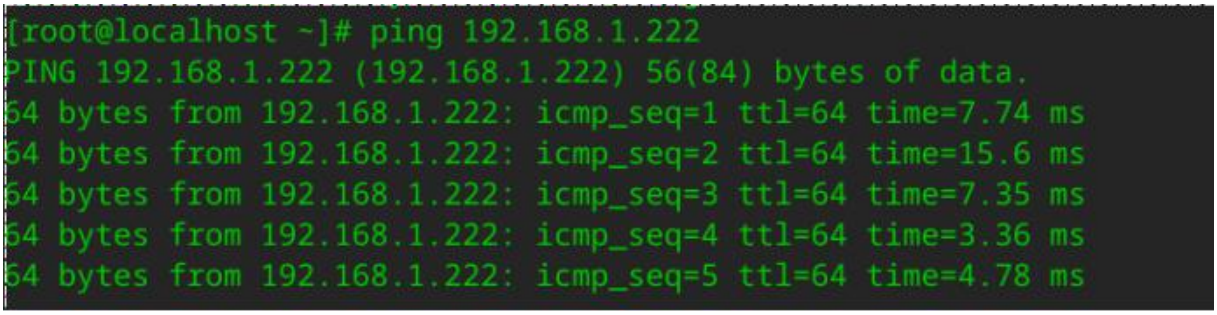
```
#systemctl stop firewalld
#systemctl disable firewalld
```

配置虚拟机 IP

```
#ip addr add 192.168.1.111/24 dev ens2 (Host1 虚拟机)
#ip addr add 192.168.1.222/24 dev ens2 (Host2 虚拟机)
```

Host1 主机的虚拟机内验证是否和 Host2 主机内的虚拟机连通

```
#ping 192.168.1.222
```



7.1.5. 配置 vlan 组网

vlan 网络也是 openstack 项目中租户常用的网络类型。ovs 交换机可以实现 vlan 隔离，功能上类似于普通交换的 vlan 隔离。vlan 隔离在 openstack 的网络中发挥着十分重要的作用。ovs 的隔离通过 tag 标签来实现。在主机 Host1 中利用 ovs 模拟租户 vlan 网络隔离，案例如下图所示：

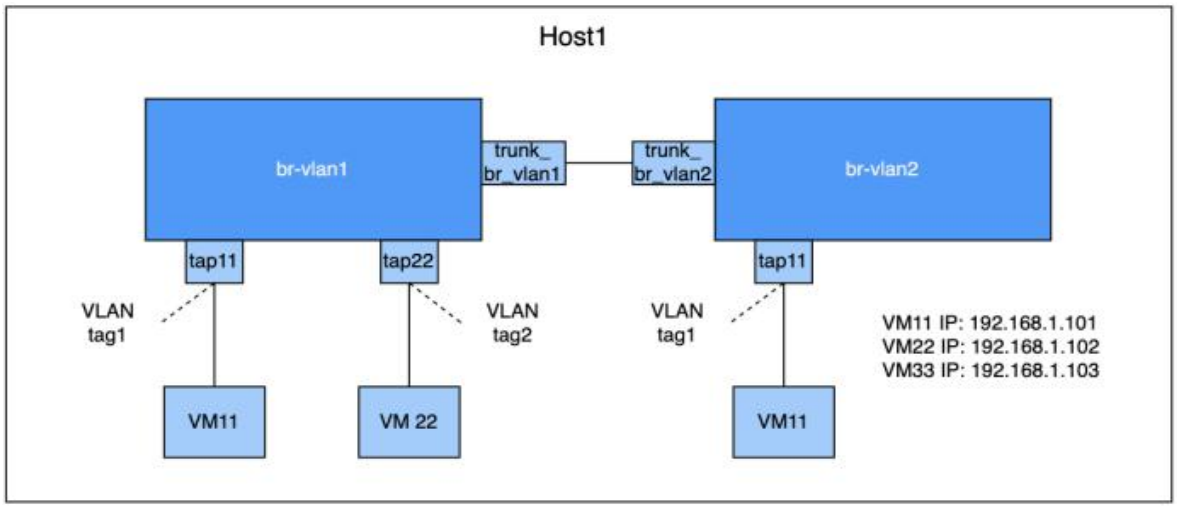


图 7-2 vlan 组网

7.1.5.1. 配置单网桥 vlan 隔离

1)创建网桥 br-vlan1，添加 dpdkvhostuserclient 端口 tap11 和 tap22

```
    #ovs-vsctl add-br br-vlan1 -- set bridge br-vlan1 datapath_type=netdev
    #ovs-vsctl add-port br-vlan1 tap11 -- set Interface tap11
type=dpdkvhostuserclient
options:vhost-server-path=/var/run/openvswitch/tap11
    #ovs-vsctl add-port br-vlan1 tap22 -- set Interface tap22
type=dpdkvhostuserclient
options:vhost-server-path=/var/run/openvswitch/tap22
```

2)将 tap11 和 tap22 加入不同的 vlan 中

```
    #ovs-vsctl set Port tap11 tag=1
    #ovs-vsctl set Port tap22 tag=2
```

3)启动虚拟机 VM11 和 VM22

虚拟机配置参照上一节，注意修改虚拟机 xml 中 Interface 的 source。

4)验证同网桥同网段不同 vlan 网络的连通性

进入虚拟机关闭防火墙

```
    #systemctl stop firewalld
    #systemctl disable firewalld
```

配置虚拟机 IP

```
    #ip addr add 192.168.1.101/24 dev ens2 (VM11)
    #ip addr add 192.168.1.102/24 dev ens2 (VM22)
```

在 VM11 和 VM22 是否连通，不连通代表隔离成功

```
#ping 192.168.1.102
```

7.1.5.2. 配置跨网桥 vlan 隔离

1)创建网桥 br-vlan2，添加 dpdkvhostuserclient 端口 tap33

```
#ovs-vsctl add-br br-vlan2 -- set bridge br-vlan2 datapath_type=netdev
#ovs-vsctl add-port br-vlan2 tap33 -- set Interface tap33
type=dpdkvhostuserclient
options:vhost-server-path=/var/run/openvswitch/tap33
```

2)将 tap33 加入 tap11 所属的 vlan 中

```
#ovs-vsctl set Port tap33 tag=1
```

3)创建 patch 口连接网桥 br-vlan1 和 br-vlan2

```
#ovs-vsctl add-port br-vlan1 trunk_br_vlan1 -- set Interface
trunk_br_vlan1 type=patch options:peer=trunk_br_vlan2
#ovs-vsctl add-port br-vlan2 trunk_br_vlan2 -- set Interface
trunk_br_vlan2 type=patch options:peer=trunk_br_vlan1
```

4)启动虚拟机 VM33

5)验证虚拟机 VM11 和 VM33 的连通性

虚拟机连通，跨网桥同 vlan 虚拟机通信成功。

6)将两网桥 patch 设置为 trunk，且只允许 vlan 2 通过，vlan1 无法通过

```
#ovs-vsctl set port trunk_br_vlan1 VLAN_mode=trunk
#ovs-vsctl set port trunk_br_vlan2 VLAN_mode=trunk
#ovs-vsctl set port trunk_br_vlan1 trunk=2
#ovs-vsctl set port trunk_br_vlan2 trunk=2
```

7)验证虚拟机 VM11 和 VM33 的连通性

虚拟机不连通，跨网桥同 vlan 网络隔离成功。

7.2. 智能网卡 ovs 硬件卸载

7.2.1. 概述

基于 OVS 软件的解决方案占用大量 CPU，会影响系统性能并阻碍可用带宽的充分利用。NVIDIA 加速交换和数据包处理（ASAP2）技术允许通过在 ConnectX-5 以后的 NIC 硬件（嵌入式交换机或 eSwitch）中处理 OVS 数据平面来卸载 OVS，同时保

持 OVS 控制平面不变。

7.2.2. 驱动安装

1. 驱动下载

在 https://network.nvidia.com/products/infiniband-drivers/linux/mlnx_ofed/下对应系统驱动。

2. 解压驱动压缩包，在驱动文件夹下使用如下命令进行编译和安装：

```
#./mlnxofedinstall --add-kernel-support --distro <cat distro 命令返回值>
```

3. 根据命令执行结果提示安装需要的软件包并执行如下命令后进行重启

```
# dracut -f
#/etc/init.d/openidb restart
```

7.2.3. 基础环境配置

1. 在/etc/default/grub 文件的“GRUB_CMDLINE_LINUX”添加 IOMMU 参数

海光机器添加：

```
#amd_iommu=on iommu=pt
```

arm 机器添加：

```
#iommu.passthrough=1
```

UEFI 启动方式写配置：

```
#grub2-mkconfig -o /boot/efi/EFI/kylin/grub.cfg
```

BIOS 启动方式写配置：

```
#grub2-mkconfig -o /boot/grub2/grub.cfg
```

2. 关闭 selinux 和防火墙

```
#setenforce 0
#systemctl stop firewalld
```

3. 网卡固件开启 SRIOV 功能

```
#mst start
```

```
#mst status
```

7.2.4. 确认 Mellanox 网卡相关信息

1. 查看 Mellanox 网卡是否存在

```
# lspci | grep Mellanox
01:00.0 Ethernet controller: Mellanox Technologies MT27800 Family [ConnectX-5]
01:00.1 Ethernet controller: Mellanox Technologies MT27800 Family [ConnectX-5]
```

2. 查看 Mellanox 网卡端口信息

```
# ls -l /sys/class/net/ | grep 01:00.0
lrwxrwxrwx 1 root root 0  9 月 11 14:19 enp1s0f0np0 -> ../../devices/pci0000:00/0000:00:00.0/0000:01:00.0/net/enp1s0f0np0
```

3. 查看 Mellanox 网卡支持的最大 vf 数

```
# cat /sys/class/net/enp1s0f0np0/device/sriov_totalvfs
8
```

7.2.5. 配置内核态 SR-IOV

1. 给 pf 网口添加 vf

```
# echo 2 > /sys/class/net/<网卡名称>/device/sriov_numvfs
```

2. 配置 vf 端口 MAC 地址

```
# ip link set <网卡名称> 0 vf 0 mac <任意 mac 地址>
```

3. 解绑 VF

```
# echo <vf pci> /sys/bus/pci/drivers/<mlx 驱动>/unbind
```

4. 切换网口模式

将 PF 设备上的“eSwitch”模式从“Legacy”修改为“SwitchDev”

```
# devlink dev eswitch set pci/<网卡 pci> mode switchdev
# echo switchdev > /sys/class/net/<网卡名称>/compat/devlink/mode
```

5. 绑定 VF

```
# echo <vf pci> /sys/bus/pci/drivers/<mlx 驱动>/bind
```

7.2.6. ovs 组网配置

1. ovs 服务启动

```
# yum install openvswitch
# systemctl start openvswitch
```

2. 使能卸载

```
# ovs-vsctl set Open_vSwitch . other-config:hw-offload=true
# ovs-vsctl set Open_vSwitch . other-config:tc-policy=verbose
```

3. 重启 ovs

```
# systemctl restart openvswitch
```

4. 创建组网

```
# ovs-vsctl add-br ovs-sriov
# ovs-vsctl add-port ovs-sriov <pf 网卡名称>
# ovs-vsctl add-port ovs-sriov <vf 网卡名称>
# ip link set dev <pf 网卡名称> up
# ip link set dev <vf 网卡名称> up
```

7.2.7. 创建虚拟机验证

1. 虚拟机配置修改

创建两个虚拟机 vm1 和 vm2

```
# virsh edit vm1
删除 interface 配置部分，举例如下：
   <interface type='network'>
     <mac address='52:54:00:f1:8b:c9'/>
     <source network='default'/>
     <model type='e1000'/>
     <address type='pci' domain='0x0000' bus='0x00' slot='0x03'
function='0x0'/>
   </interface>
替换为：
   <hostdev mode='subsystem' type='pci' managed='yes'>
     <source>
       <address domain='0x0000' bus='0x01' slot='0x00' function='0x3'/>
     </source>
   </hostdev>
其中 <address domain='0x0000' bus='0x01' slot='0x00' function='0x3'/>
中 domain、bus、slot、function 对应 vf 的 pci 端口号为 0000:01:03:0
```

vm2 使用不同的 vf，重复 vm1 修改配置操作。

2. 验证连通性

关闭两个虚拟机的 selinux 和防火墙

```
# setenforce 0
# systemctl stop firewalld
```

3. 配置 ip

```
# ip addr add 10.1.1.100/24 dev ens3  #vm1 ip 和网卡名称可根据实际情况修改
# ip addr add 10.1.1.200/24 dev  #vm2 ens3
```

在 vm1 中 ping vm2 验证连通性

7.3. cilium 部署验证

7.3.1. 概述

cilium 是一个开源的云原生解决方案，用于提供、保护（安全功能）和观察（监控功能）工作负载之间的网络连接，由革命性的内核技术 eBPF 提供动力。cilium 主要使用场景在 kubernetes 中。在 kubernetes 环境中，cilium 可充当网络插件，提供 pod 之间的连接。它通过执行网络策略(network policy)和透明加密来提供安全性，而 cilium 的 hubble 组件则提供了网络流量的深度可见性（监控功能）。

7.3.2. k8s 集群部署

k8s 集群部署参照 7.2 节，集群至少两个节点。

7.3.3. cilium 网络插件部署

1. 环境配置

物理机/虚拟机环境均可，可通外网。

2. 软件源配置

```
# cat /etc/yum.repos.d/kylin_x86_64.repo
...
 [v10-hostos-2309-epkl]
name = v10-hostos-2309-epkl
```

```
baseurl = https://update.cs2c.com.cn/NS/HOST/2309/epkl/$basearch/
enable =1
...
```

3. cilium-chart 文件下载

cilium 1.12.11 和 1.13.4 版本上游仓库的 yaml 文件已上传到 gitee 仓库中。

```
# wget
https://gitee.com/mengxuanzhang/cilium/repository/archive/master.zip
# unzip cilium-master.zip
# cd cilium-master/cilium-master
# ls
```

```
cilium-chart-1.12.11  cilium-chart-1.13.4  README.en.md  README.md
```

将 cilium-chart-1.12.11 和 cilium-chart-1.13.4 两个文件上传到 master 节点的 /home 目录下。

4. cilium 安装

```
# yum install cilium-cli
# cilium install --chart-directory /home/cilium-chart-<版本号>
# cilium status
```

```
[root@node ~]# cilium status
    /¯¯\
 /¯¯\__/¯¯\    Cilium:          OK
 \__/¯¯\__/    Operator:        OK
 /¯¯\__/¯¯\    Hubble Relay:    disabled
 \__/¯¯\__/    ClusterMesh:     disabled
    \__/

Deployment        cilium-operator    Desired: 1, Ready: 1/1, Available: 1/1
DaemonSet         cilium             Desired: 2, Ready: 2/2, Available: 2/2
Containers:       cilium             Running: 2
                  cilium-operator    Running: 1
Cluster Pods:     7/7 managed by Cilium
Image versions    cilium-operator    quay.io/cilium/operator-generic:v1.12.11: 1
                  cilium             quay.io/cilium/cilium:v1.12.11: 2
```

7.3.4. hubble 插件部署

- 安装 hubble

```
# cilium hubble enable --chart-directory /home/cilium-chart-<版本号>
```

- hubble api 端口配置

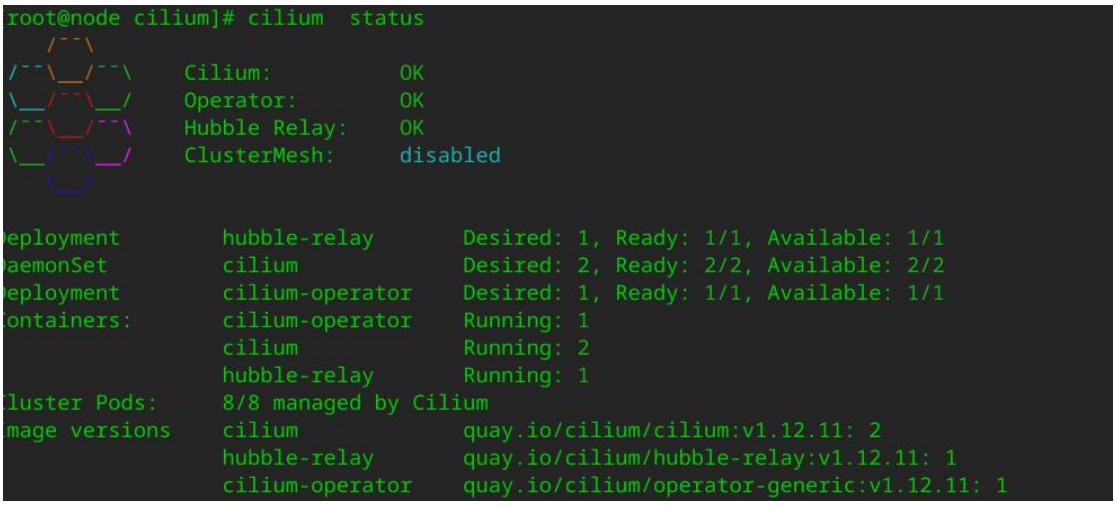
```
# cilium hubble port-forward&
```

7.3.5. 功能验证

7.3.5.1. cilium 集群状态查看

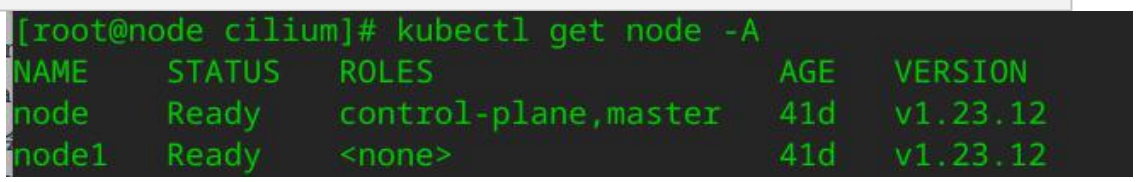
1. 查看 cilium 状态

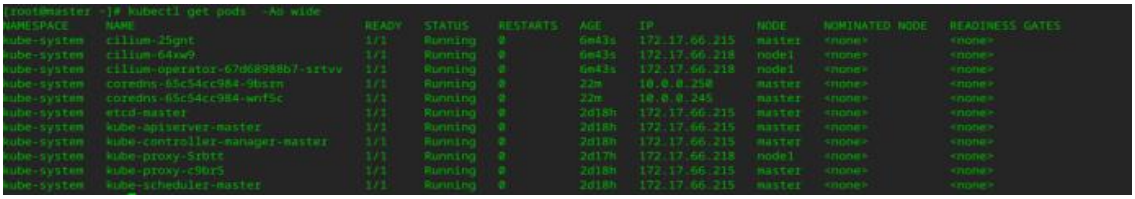
```
# cilium status
```



2. 查看 node 和 pod 的状态

```
# kubectl get node -A
# kubectl get pods -A
```





3. 查看 cilium-agent 的功能开启状态

```
# kubectl -n kube-system exec ds/cilium -- cilium status
```

4. 查看集群的 service ip

```
# kubectl get svc -A
```

```
[root@node ~]# kubectl get svc -A
NAMESPACE     NAME           TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)                  AGE
default       kubernetes     ClusterIP   10.96.0.1      <none>        443/TCP                  41d
kube-system   hubble-peer    ClusterIP   10.96.0.188    <none>        443/TCP                  12h
kube-system   hubble-relay   ClusterIP   10.96.0.16     <none>        80/TCP                   12h
kube-system   kube-dns       ClusterIP   10.96.0.10     <none>        53/UDP,53/TCP,9153/TCP   41d
```

5. 查看 hubble 状态

```
# hubble status
```

```
[root@node cilium]# hubble status
Healthcheck (via localhost:4245): Ok
Current/Max Flows: 8,190/8,190 (100.00%)
Flows/s: 5.30
Connected Nodes: 2/2
```

6. hubble 流量监控

```
# hubble observe
```

```
[root@node cilium]# hubble observe
Sep 12 03:25:51.451: 10.0.1.239:34678 (remote-node) <> 10.0.0.5:4240 (health) to-overlay FORWARDED (TCP Flags: ACK, PSH)
Sep 12 03:25:51.452: 10.0.1.239:34678 (remote-node) <> 10.0.0.5:4240 (health) to-overlay FORWARDED (TCP Flags: ACK)
Sep 12 03:25:51.695: 10.0.1.239:53462 (host) -> kube-system/coredns-65c54cc984-q7qfv:8080 (ID:58341) to-endpoint FORWARDED (TCP Flags: SYN)
Sep 12 03:25:51.695: 10.0.1.239:53462 (host) <- kube-system/coredns-65c54cc984-q7qfv:8080 (ID:58341) to-stack FORWARDED (TCP Flags: SYN, ACK)
Sep 12 03:25:51.695: 10.0.1.239:52358 (host) -> kube-system/coredns-65c54cc984-q7qfv:8181 (ID:58341) to-endpoint FORWARDED (TCP Flags: SYN)
Sep 12 03:25:51.695: 10.0.1.239:52358 (host) <- kube-system/coredns-65c54cc984-q7qfv:8181 (ID:58341) to-stack FORWARDED (TCP Flags: SYN, ACK)
Sep 12 03:25:51.695: 10.0.1.239:53462 (host) -> kube-system/coredns-65c54cc984-q7qfv:8080 (ID:58341) to-endpoint FORWARDED (TCP Flags: ACK)
Sep 12 03:25:51.695: 10.0.1.239:52358 (host) -> kube-system/coredns-65c54cc984-q7qfv:8181 (ID:58341) to-endpoint FORWARDED (TCP Flags: ACK)
Sep 12 03:25:51.695: 10.0.1.239:52358 (host) -> kube-system/coredns-65c54cc984-q7qfv:8181 (ID:58341) to-endpoint FORWARDED (TCP Flags: ACK, PSH)
```

7.3.5.2. cilium 网络连通性验证

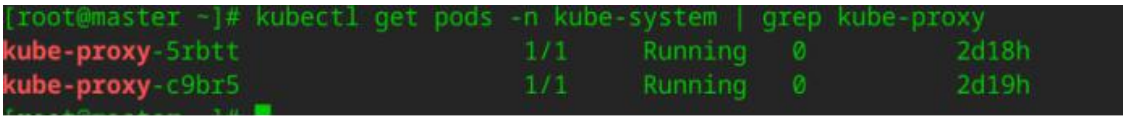
1. cilium kubeProxyReplacement 配置

cilium 提供了一种称为 "kube-proxy 替代"（kubeProxyReplacement）的功能，旨在改进 Kubernetes 群集中的网络性能和安全性。kube-proxy 替代功能可以显著提高网络性能。传统的 kube-proxy 是一个基于 iptables 的代理，用于处理 Kubernetes 中的服务发现和负载均衡。但随着容器数量的增加，这种代理模式可能导致性能下降。Cilium 的 kube-proxy 替代功能使用了更高效的技术，如 eBPF，以加速服务发现和负载均衡操作。

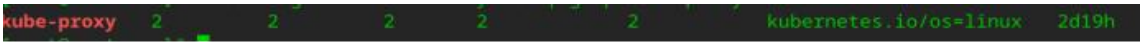
- 首先检查一下 cilium 状态，可以看到 KubeProxyReplacement 参数默认是设置为 Disabled。再检查一下集群中的 kube-proxy 组件状态，看到各个组件都工作正常。

```
# kubectl -n kube-system exec ds/cilium -- cilium status | grep KubeProxyReplacement
```


```
# kubectl get pods -n kube-system | grep kube-proxy
```




```
# kubectl get ds -n kube-system | grep kube-proxy
```




```
# kubectl get cm -n kube-system | grep kube-proxy
```

- kube-proxy 配置备份

删除 kube-proxy 之前，先备份下相关配置， 在 master 节点上备份 kube-proxy 相关的配置。

```
# kubectl get ds -n kube-system kube-proxy -o yaml > kube-proxy-ds.yaml
```

```
# kubectl get cm -n kube-system kube-proxy -o yaml > kube-proxy-cm.yaml
```

在每台机器上面使用 root 权限备份一下 iptables 规则

```
# iptables-save > kube-proxy-iptables-save.bak
```

- kube-proxy 配置删除

删除 daemonset kube-proxy

```
# kubectl -n kube-system delete ds kube-proxy
```

删除掉 kube-proxy 的 configmap，防止以后使用 kubeadm 升级 k8s 的时候重新安装 kube-proxy

```
# kubectl -n kube-system delete cm kube-proxy
```

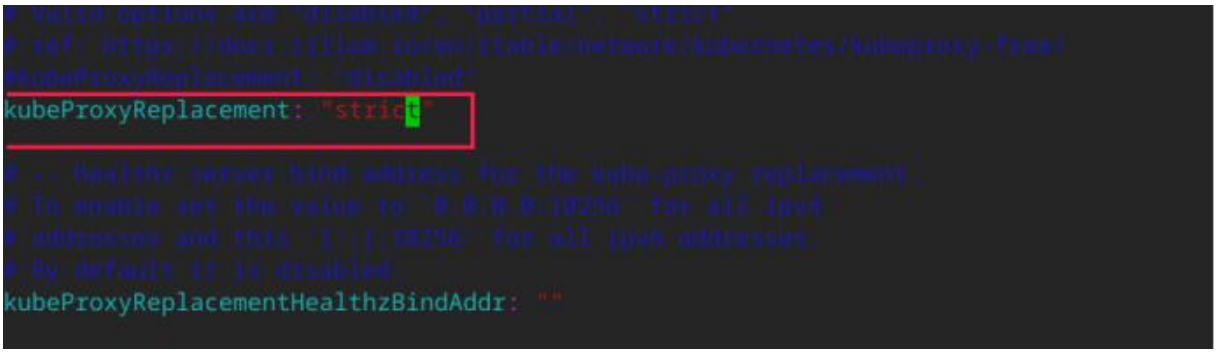
卸载 cilium

```
# cilium uninstall
```

- 开启 cilium 的 kube-proxy-replacement 功能来替代 kube-proxy

通过修改 cilium-configmap 的方式开启 kube-proxy-replacement，将 kubeProxyReplacement: “disabled”修改为”strict”

```
# vim /home/cilium-chart-1.13.4/values.yaml
```



重新启动 cilium

```
# cilium install --chart-directory /home/cilium-chart-1.13.4
# kubectl -n kube-system exec ds/cilium -- cilium status --verbose
```

```
KubeProxyReplacement Details:
  Status:                 Strict
  Socket LB:              Enabled
  Socket LB Coverage:     Full
  Socket LB Protocols:    TCP, UDP
  Devices:                ens3 172.17.66.168, ens8 192.168.122.238
  Mode:                   SNAT
  Backend Selection:      Random
  Session Affinity:       Enabled
  Graceful Termination:   Enabled
  NAT46/64 Support:       Disabled
  XDP Acceleration:       Disabled
  Services:
  - ClusterIP:      Enabled
  - NodePort:       Enabled (Range: 30000-32767)
  - LoadBalancer:   Enabled
  - externalIPs:    Enabled
  - HostPort:       Enabled
```

开启 KubeProxyReplacement 后 Socket LB（套接字级别负载均衡器绕过功能：在连接（TCP、已连接 UDP）、sendmsg（UDP）或 recvmsg（UDP）系统调用时，通过检查目标 IP 是否存在于服务 IP 中，并选择一个服务后端作为目标，套接字级别负载均衡器在 Cilium 的底层数据路径上进行透明操作。这意味着尽管应用程序假设它连接到了服务地址，但相应的内核套接字实际上是连接到了后端地址，因此不需要额外的底层 NAT）、Session Affinity（会话亲和性功能：允许在 Cilium 中配置的 Kubernetes 服务中，来自同一 Pod 或主机的连接始终选择相同的服务终端，以实现流量的负载均衡和会话保持）等等都自动开启。

2. 连通性验证

- 创建测试 pod

场景：以下的 nginx.yaml 配置清单包含了一组资源，用于部署一个简单的 nginx 服务以及一个客户端应用程序，客户端应用程序使用 tcpreplay 发送流量到 nginx 服务。

➢ 部署名为 nginx1 和 nginx2 的两个 Nginx Deployment。每个 Deployment 包含一个 Pod，Pod 中运行 nginx 容器。这两个 Deployment 都在同一个 Kubernetes 节点上运行（nodeSelector 部分指定了主节点，这需要集群有多个节点并且有一个主节点）。

➢ 创建一个名为 test 的 Service，该 Service 将流量路由到带有 app: nginx 标签的 Pod，这两个 Nginx Deployment 都有这个标签。

➢ 创建名为 client 的 Deployment，其中包含一个 Pod，Pod 中运行 dgarros/tcpreplay:latest 镜像的容器。这个容器用于发送流量到 Nginx 服务。这个 Deployment 在名为 node1 的 Kubernetes 节点上运行（同样使用 nodeSelector 指定）。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx1
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      nodeSelector:
        kubernetes.io/hostname: node
      containers:
        - name: hello
          image: nginx
          imagePullPolicy: IfNotPresent
          ports:
            - name: http
              containerPort: 80

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx2
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      nodeSelector:
        kubernetes.io/hostname: node
      containers:
        - name: hello
          image: nginx
```

```
        imagePullPolicy: IfNotPresent
        ports:
          - name: http
            containerPort: 80

---
apiVersion: v1
kind: Service
metadata:
  name: test
spec:
  selector:
    app: nginx
  ports:
  - protocol: TCP
    port: 9999
    targetPort: 80

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: client
spec:
  selector:
    matchLabels:
      app: test
  replicas: 1
  template:
    metadata:
      labels:
        app: test
    spec:
      nodeSelector:
        kubernetes.io/hostname: node1
      containers:
        - name: hello
          image: dgarros/tcpreplay:latest
        imagePullPolicy: IfNotPresent
```

应用 nginx.yaml，创建测试 pod

```
# kubectl apply -f nginx.yaml
```

- 查看 service ip

```
# kubectl get svc -Ao wide
```

```
[root@node cilium]# kubectl get svc -Ao wide
NAMESPACE     NAME           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)                  AGE    SELECTOR
default       kubernetes     ClusterIP   10.96.0.1       <none>        443/TCP                  41d    <none>
default       test           ClusterIP   10.96.0.151     <none>        9999/TCP                 2m28s  app=nginx
kube-system   hubble-peer    ClusterIP   10.96.0.188     <none>        443/TCP                  14h    k8s-app=cilium
kube-system   hubble-relay   ClusterIP   10.96.0.16      <none>        80/TCP                   14h    k8s-app=hubble-relay
kube-system   kube-dns       ClusterIP   10.96.0.10      <none>        53/UDP,53/TCP,9153/TCP   41d    k8s-app=kube-dns
```

```
# kubectl get pods -Ao wide
```

```
[root@node cilium]# kubectl get pod -Ao wide
NAMESPACE     NAME                              READY   STATUS    RESTARTS       AGE    IP             NODE    NOMINATED NODE   READINESS G
ATES
default       client-7c8444b54d-5fqm6           1/1     Running   0              2m2s   10.0.0.156     node1   <none>           <none>
default       nginx1-779c8664c8-9jxwx           1/1     Running   0              2m2s   10.0.1.8       node    <none>           <none>
default       nginx2-779c8664c8-ts22c           1/1     Running   0              2m2s   10.0.1.76      node    <none>           <none>
```

- pod 与 node 间连通验证

在任意 node 上 ping 三个新建 pod 的 ip

```
# ping 10.0.0.156
# ping 10.0.1.8
# ping 10.0.1.76
```

```
[root@node cilium]# ping 10.0.1.8
PING 10.0.1.8 (10.0.1.8) 56(84) 字节的数据。
64 字节，来自 10.0.1.8: icmp_seq=1 ttl=63 时间=0.102 毫秒
64 字节，来自 10.0.1.8: icmp_seq=2 ttl=63 时间=0.040 毫秒
^C
--- 10.0.1.8 ping 统计 ---
已发送 2 个包， 已接收 2 个包，0% packet loss, time 1009ms
rtt min/avg/max/mdev = 0.040/0.071/0.102/0.031 ms
[root@node cilium]# ping 10.0.0.156
PING 10.0.0.156 (10.0.0.156) 56(84) 字节的数据。
64 字节，来自 10.0.0.156: icmp_seq=1 ttl=63 时间=0.666 毫秒
64 字节，来自 10.0.0.156: icmp_seq=2 ttl=63 时间=0.631 毫秒
^C
--- 10.0.0.156 ping 统计 ---
已发送 2 个包， 已接收 2 个包，0% packet loss, time 1019ms
rtt min/avg/max/mdev = 0.631/0.648/0.666/0.017 ms
[root@node cilium]# ping 10.0.1.76
PING 10.0.1.76 (10.0.1.76) 56(84) 字节的数据。
64 字节，来自 10.0.1.76: icmp_seq=1 ttl=63 时间=0.147 毫秒
64 字节，来自 10.0.1.76: icmp_seq=2 ttl=63 时间=0.046 毫秒
64 字节，来自 10.0.1.76: icmp_seq=3 ttl=63 时间=0.050 毫秒
^C
--- 10.0.1.76 ping 统计 ---
已发送 3 个包， 已接收 3 个包，0% packet loss, time 2075ms
rtt min/avg/max/mdev = 0.046/0.081/0.147/0.046 ms
```

- pod 与 pod 间连通验证

```
# kubectl exec -it -n default  client-7c8444b54d-5fqm6  -- ping 10.0.1.8
# kubectl exec -it -n default   client-7c8444b54d-5fqm6  -- ping 10.0.1.76
```

```
[root@node cilium]# kubectl exec -it -n default       client-7c8444b54d-5fqm6  -- ping 10.0.1.8
PING 10.0.1.8 (10.0.1.8) 56(84) bytes of data.
64 bytes from 10.0.1.8: icmp_seq=1 ttl=63 time=0.534 ms
64 bytes from 10.0.1.8: icmp_seq=2 ttl=63 time=0.432 ms
^C
--- 10.0.1.8 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1052ms
rtt min/avg/max/mdev = 0.432/0.483/0.534/0.051 ms
[root@node cilium]# kubectl exec -it -n default       client-7c8444b54d-5fqm6  -- ping 10.0.1.76
PING 10.0.1.76 (10.0.1.76) 56(84) bytes of data.
64 bytes from 10.0.1.76: icmp_seq=1 ttl=63 time=0.587 ms
64 bytes from 10.0.1.76: icmp_seq=2 ttl=63 time=0.492 ms
64 bytes from 10.0.1.76: icmp_seq=3 ttl=63 time=0.407 ms
^C
--- 10.0.1.76 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2036ms
rtt min/avg/max/mdev = 0.407/0.495/0.587/0.075 ms
```

- pod 与 service 间连通验证

client pod 发起对 service 的访问

```
# kubectl exec -it -n default  client-7c8444b54d-5fqm6  -- curl http://10.96.0.151:9999
```

开启多个 node 节点终端，抓取 10.0.1.8 和 10.0.1.76 的流量，可发现进入 service ip 的流量会直接负载到后端的 pod ip 上

```
# tcpdump -i any host 10.0.1.8
# tcpdump -i any host 10.0.1.76
```

```
1:13:18.598803 cilium_vxlan Out IP 10.0.1.8.http > 10.0.0.156.36108: Flags [P.], seq 239:854, ack 81, win 502, options [nop,nop,TS val 26887
01373 ecr 4069064666], length 615: HTTP
1:13:18.599162 cilium_vxlan P   IP 10.0.0.156.36108 > 10.0.1.8.http: Flags [.], ack 239, win 506, options [nop,nop,TS val 4069064667 ecr 268
8701373], length 0
1:13:18.599300 cilium_vxlan P   IP 10.0.0.156.36108 > 10.0.1.8.http: Flags [.], ack 854, win 502, options [nop,nop,TS val 4069064667 ecr 268
8701373], length 0
1:13:18.599438 cilium_vxlan P   IP 10.0.0.156.36108 > 10.0.1.8.http: Flags [F.], seq 81, ack 854, win 502, options [nop,nop,TS val 406906466
7 ecr 2688701373], length 0
1:13:18.599515 lxc8c6391952c39 In  IP 10.0.1.8.http > 10.0.0.156.36108: Flags [F.], seq 854, ack 82, win 502, options [nop,nop,TS val 268870
1374 ecr 4069064667], length 0
1:13:18.599528 cilium_vxlan Out IP 10.0.1.8.http > 10.0.0.156.36108: Flags [F.], seq 854, ack 82, win 502, options [nop,nop,TS val 268870137
4 ecr 4069064667], length 0
1:13:18.600114 cilium_vxlan P   IP 10.0.0.156.36108 > 10.0.1.8.http: Flags [.], ack 855, win 502, options [nop,nop,TS val 4069064668 ecr 268
8701374], length 0
```

- cilium connectivity test

cilium 连接性测试旨在确保 Cilium 能够在不同情况下正常连接和保护容器和微服务。这些连接性测试只能在一个没有其他 Pod 或网络策略的命名空间中运行。如果启用了 Cilium Clusterwide Network Policy，这可能会干扰连接性检查。这是因为网络策略和 Cilium Clusterwide Network Policy 可能会干扰测试中期望的通信行为。在主节点上执行:

```
# cilium connectivity test
```

7.4. gazelle 部署使用

7.4.1. 概述

Gazelle 是一款高性能用户态协议栈。它基于 DPDK 在用户态直接读写网卡报文，共享大页内存传递报文，使用轻量级 LwIP 协议栈。能够大幅提高应用的网络 I/O 吞吐能力。

7.4.2. gazelle 部署

1. 将网卡绑定到 dpdk 用户态

a.安装 NIC 网卡驱动模块 vfio-pci

```
#modprobe vfio-pci
#chmod a+x /dev/vfio
#chmod 0666 /dev/vfio/*
```

注：如果系统没有 IOMMU，启用 noiommu 模式，echo Y > /sys/module/vfio/parameters/enable_unsafe_noiommu_mode

b.查看网口 pci 信息，找到需要绑定网口的 PCI 地址

```
#dpdk-devbind -s
```

c.执行绑定

```
#dpdk-devbind --bind=vfio-pci 0000:00:08.0
```

注：0000:00:08.0 为网卡 pci 值，请根据实际情况填写

2. 大页配置

a. 大页内存配置

Gazelle 使用大页内存提高效率。使用 root 权限配置系统预留大页内存，可选用任意页大小。因每页内存都需要一个 fd，使用内存较大时，建议使用 1G 的大页，避免占用过多 fd。x86 环境可选择的大页为 2M 或 1G 大小，aarch64 环境可选择的大页为 2M 或 512M 大小。配置命令如下：

```
#配置 1G 大页内存：在 node0 上配置 1G * 5 = 5G
echo 5 >
```

```
/sys/devices/system/node/node0/hugepages/hugepages-1048576kB/nr_hugepages
```

b. 挂载大页内存

```
    # mkdir -p /mnt/hugepages-ltran
    # mkdir -p /mnt/hugepages-lstack
    # chmod -R 700 /mnt/hugepages-ltran
    # chmod -R 700 /mnt/hugepages-lstack
    # 注: /mnt/hugepages-ltran 和 /mnt/hugepages-lstack 必须挂载同样 pagesize 的大页内存。
    # mount -t hugetlbfs nodev /mnt/hugepages-ltran -o pagesize=1G
    # mount -t hugetlbfs nodev /mnt/hugepages-lstack -o pagesize=1G
```

3. gazelle 配置文件

lstack.conf 用于指定 lstack 的启动参数，默认路径为/etc/gazelle/lstack.conf, 配置文件参数如下:

选项	参数格式	说明
dpdk_args	--socket-mem（必需） --huge-dir（必需） --proc-type（必需） --legacy-mem --map-perfect -d 等	dpdk 初始化参数，参考 dpdk 说明 对于没有链接到 liblstack.so 的 PMD，必须使用 -d 加载，比如 librte_net_mlx5.so。
use_ltran	0/1	是否使用 ltran
listen_shadow	0/1	是否使用影子 fd 监听，单个 listen 线程多个协议栈线程时使用
num_cpus	"0,2,4 ..."	lstack 线程绑定的 cpu 编号，编号的数量为 lstack 线程个数(小于等于网卡多队列数量)。可按 NUMA 选择 cpu

选项	参数格式	说明
app_bind_numa	0/1	应用的 epoll 和 poll 线程是否绑定到协议栈所在的 numa，缺省值是 1，即绑定
app_exclude_cpus	"7,8,9 ..."	应用的 epoll 和 poll 线程不会绑定到的 cpu 编号，app_bind_numa = 1 时才生效
low_power_mode	0/1	是否开启低功耗模式，暂不支持
kni_swith	0/1	rte_kni 开关，默认为 0。只有不使用 ltran 时才能开启
unix_prefix	"string"	gazelle 进程间通信使用的 unix socket 文件前缀字符串，默认为空，和需要通信的 ltran.conf 的 unix_prefix 或 gazellectl 的-u 参数配置一致。不能含有特殊字符，最大长度为 128。
host_addr	"192.168.xx.xx"	协议栈的 IP 地址，必须和 redis-server 配置文件里的“bind”字段保存一致。
mask_addr	"255.255.xx.xx"	掩码地址
gateway_addr	"192.168.xx.1"	网关地址
devices	"aa:bb:cc:dd:ee:ff"	网卡通信的 mac 地址，需要与 ltran.conf 的 bond_macs 配置一致；在 lstack bond1 模式下，可指定 bond1 的主接口，取值为 bond_slave_mac 之一

选项	参数格式	说明
send_connect_number	4	设置为正整数，表示每次协议栈循环中发包处理的连接个数
read_connect_number	4	设置为正整数，表示每次协议栈循环中收包处理的连接个数
rpc_number	4	设置为正整数，表示每次协议栈循环中 rpc 消息处理的个数
nic_read_num	128	设置为正整数，表示每次协议栈循环中从网卡读取的数据包的个数
tcp_conn_count	1500	tcp 的最大连接数，该参数乘以 mbuf_count_per_conn 是初始化时申请的 mbuf 池大小，配置过小会启动失败，tcp_conn_count * mbuf_count_per_conn * 2048 字节不能大于大页大小
mbuf_count_per_conn	170	每个 tcp 连接需要的 mbuf 个数，该参数乘以 tcp_conn_count 是初始化时申请的 mbuf 地址池大小，配置过小会启动失败，tcp_conn_count * mbuf_count_per_conn * 2048 字节不能大于大页大小
nic_rxqueue_size	4096	网卡接收队列深度，范围 512-8192，缺省值是 4096
nic_txqueue_size	2048	网卡发送队列深度，范围

选项	参数格式	说明
		512-8192，缺省值是 2048
nic_vlan_mode	-1	vlan 模式开关，变量值为 vlanid，取值范围-1~4094，-1 关闭，缺省值是-1
bond_mode	n	bond 模式，目前支持 ACTIVE_BACKUP/8023AD/ALB 三种模式，对应的取值是 1/4/6；当取值为-1 或者 NULL 时，表示未配置 bond
bond_slave_mac	"aa:bb:cc:dd:ee:ff;dd:aa:cc:dd:ee:ff"	用于组 bond 的两个子口的 mac 地址
bond_miimon	n	链路监控时间，单位为 ms，取值范围为 1 到 2^31 - 1，缺省值为 10ms

lstack.conf 示例：

```
    dpdk_args=["--socket-mem", "2048,0,0,0", "--huge-dir",
"/mnt/hugepages-lstack", "--proc-type", "primary", "--legacy-mem",
"--map-perfect"]
    use_ltran=1
    kni_switch=0
    low_power_mode=0
    num_cpus="2,22"
    host_addr="192.168.1.10"
    mask_addr="255.255.255.0"
    gateway_addr="192.168.1.1"
    devices="aa:bb:cc:dd:ee:ff"
    send_connect_number=4
    read_connect_number=4
    rpc_number=4
    nic_read_num=128
    tcp_conn_count=1500
    mbuf_count_per_conn=170
```

7.4.3. gazelle 使用

1. 使用 gazelle 运行应用程序

使用 LD_RELOAD 加载 Gazelle 的库

GAZELLE_BIND_PROCNAME 环境变量指定进程名，LD_PRELOAD 指定 Gazelle 库路径。

```
    #GAZELLE_BIND_PROCNAME=test
LD_PRELOAD=/usr/lib64/liblstack.so ./test
```

2. 使用 gazellectl 进行数据监控

在 gazelle 主程序运行时，可使用 gazellectl 命令进行对 gazelle 主程序的监控。

```
    # gazellectl lstack show 1 -p TCP
```

可输出通过 gazelle 的报文流量中的 TCP 包信息。也可根据实际情况查询 UDP、IP、etharp、ICMP 等包信息。

8. 业务资源混合部署

在云上业务类型和硬件资源越来越丰富的背景下，为了提高资源利用率，云场景下让多样性业务和算力混部系统以最佳状态运行。根据业务情况，提出针对容器业务的在线离线业务混合部署，针对虚拟机的高低优先级混合部署等解决方案。

8.1. kubernetes 优先级混部

rubik 是一个部署在 kubernetes 环境集群中的一个 daemonset, 该工具通过与 kubernetes 通信，获取当前节点的 pod 信息，通过 cgroup 控制接口对其进行资源管控，保障服务质量。当前包括如下功能：

- CPU 和内存的优先级抢占
- 内存异步水位线
- blkio 限速
- ioCost 权重限速

- 潮汐亲和性控制
- 负载压力感知驱逐 pod
- 网络优先级 Qos 控制

用户使用优先级混部特性的前提条件是需要部署一个 kubernetes 集群，具体部署方法参考本手册的 kubernetes 部署部分。在部署完成 kubernetes 之后，需要使用 docker 或其他运行时加载 rubik 容器镜像，并编写 yaml 文件使用 kubectl 命令来部署 rubik。当前 rubik 支持 docker, contianerd, crio 三种运行时以及 cgroupfs，systemd 两种 cgroup driver。 rubik 进程的配置具体内容参考如下,可根据需求自行修改

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: rubik
rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["list", "watch"]
  - apiGroups: [""]
    resources: ["pods/eviction"]
    verbs: ["create"]
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: rubik
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: rubik
subjects:
  - kind: ServiceAccount
    name: rubik
    namespace: kube-system
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: rubik
  namespace: kube-system
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: rubik-config
  namespace: kube-system
data:
  config.json: |
    {
      "agent": {
        "logDriver": "stdio",
        "logDir": "/var/log/rubik",
        "logSize": 1024,
        "logLevel": "info",
        "cgroupRoot": "/sys/fs/cgroup",
        "enabledFeatures": [
          "preemption"
        ]
      },
      "preemption": {
        "resource": [
          "cpu"
```

```
        ]
      }
    }
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: rubik-agent
  namespace: kube-system
  labels:
    k8s-app: rubik-agent
spec:
  selector:
    matchLabels:
      name: rubik-agent
  template:
    metadata:
      namespace: kube-system
      labels:
        name: rubik-agent
    spec:
      serviceAccountName: rubik
      hostPID: true
      containers:
      - name: rubik-agent
        image: rubik:latest
        imagePullPolicy: IfNotPresent
        env:
          - name: RUBIK_NODE_NAME
            valueFrom:
              fieldRef:
                fieldPath: spec.nodeName
        securityContext:
          capabilities:
            add:
            - SYS_ADMIN
        resources:
          limits:
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        volumeMounts:
        - name: rubiklog
          mountPath: /var/log/rubik
          readOnly: false
        - name: runrubik
          mountPath: /run/rubik
          readOnly: false
        - name: sysfs
          mountPath: /sys/fs
          readOnly: false
```

```
    - name: devfs
      mountPath: /dev
      readOnly: false
    - name: config-volume
      mountPath: /var/lib/rubik
  terminationGracePeriodSeconds: 30
  volumes:
  - name: rubiklog
    hostPath:
      path: /var/log/rubik
  - name: runrubik
    hostPath:
      path: /run/rubik
  - name: sysfs
    hostPath:
      path: /sys/fs
  - name: devfs
    hostPath:
      path: /dev
  - name: config-volume
    configMap:
      name: rubik-config
      items:
      - key: config.json
        path: config.json
```

8.1.1. CPU 和内存的优先级抢占

rubik 支持业务优先级配置，针对在离线业务混合部署的场景，确保在线业务相对离线业务的资源抢占。目前仅支持 CPU 资源和内存资源。

内存抢占需要开启内核接口：

```
echo 1 > /proc/sys/vm/memcg_qos_enable
```

使用该特性，用户需开启 rubik 的抢占特性, 配置文件部分如下，目前仅支持 cpu 和 memory 两种类型的抢占资源。

```
...
  "agent": {
    "enabledFeatures": [
      "preemption"
    ]
  },
  "preemption": {
    "resource": [
      "cpu",
      "memory"
    ]
  }
...
```

同时, 用户需要在 pod 的 yaml 注解中增加 vocalno.sh/preemption 字段来指定业务优先级。业务优先级配置示例如下：

```
annotations
  volcano.sh/preemptable: xxx
```

该字段支持的值如下：

- always：标识为离线业务，优先级最低
- prefer：标识为离线业务，优先级次低
- normal：标识为在线任务，普通优先级
- avoid: 标识为在线业务，中高优先级
- never：标识为在线业务，高优先级
- true： 同 prefer, 向后兼容用
- false: 同 normal, 向后兼容用

通过配置该字段，高优先级业务可以抢占低优先级业务的 cpu 资源，当内存不足时，离线业务优先被 OOM-kill。举例如下：

根据如下 yaml 文件创建一个离线任务（pod 中申请的 cpu 核数为 4，具体可设置为虚拟机的最大核数）

```
apiVersion: v1
kind: Pod
metadata:
  name: "stress-offline"
  annotations:
    volcano.sh/preemptable: "always"
spec:
  containers:
    - image: "alexeiled/stress-ng:latest"
      imagePullPolicy: IfNotPresent
      name: stress-offline
      args: [ "--cpu" , "4" ]
```

再根据如下 yaml 文件创建一个在线任务

```
apiVersion: v1
kind: Pod
metadata:
  name: "stress-online"
  annotations:
    volcano.sh/preemptable: "never"
spec:
  containers:
    - image: "alexeiled/stress-ng:latest"
      imagePullPolicy: IfNotPresent
      name: stress-online
      args: [ "--cpu" , "4" ]
```

使用 kubectl apply -f 部署计算负载即可。

8.1.2. 内存异步水位线

该功能是基于内核 cgroup 的动态水位线快压制慢恢复策略。memory.high 是内核提供的 memcg 级的水位线接口，rubik 动态检测内存压力，动态调整离线应用的 memory.high 上限，实现对离线业务的内存压制，保障在线业务的服务质量。

其核心为：

- rubik 启动时计算预留内存，默认为总内存的 10%，如果总内存的 10%超过 10G，则为 10G
- 配置离线容器的 cgroup 级别水位线，内核提供 memory.high 和 memory.high_async_ratio 两个接口，分别配置 cgroup 的软上限和警戒水位线。启动 rubik 时默认配置 memory.high 为 total_memory(总内存)*80%
- 获取剩余内存 free_memory
- free_memory 小于预留内存 reserved_memory 时降低离线的 memory.high，每次降低总内存的 10%,total_memory*10%
- 持续一分钟 free_memory>2*reserved_memory 时提高离线的 memory.high，每次提升总内存的 1%，total_memory*1%

用户使用内存异步水位线需要在 rubik-daemonset.yaml 中打开对应的功能开关。

```
config.json: |
 {
  "agent": {
   "enabledFeatures": [
    "dynMemory"
   ]
  },
 "dynMemory": {
  "policy": "fssr"
 }
 }
```

8.1.3. iocost 权重限速

为了有效解决由离线业务 IO 占用过高，导致在线业务 QoS 下降的问题，rubik 容器提供了基于 cgroup v1 iocost 的 IO 权重控制功能。rubik 支持通过在 cgroup v1 下的 iocost 控制不同 Pod 的 io 权重分配。

因此需要内核支持 cgroup v1 blkcg iocost 和 cgroup v1 writeback，即在 blkcg 根系统文件下存在 blkio.cost.qos 和 blkio.cost.model 两个文件接口。

如果不存在，则需要在在开机启动项中增加 cgroup1_writeback 。

```
GRUB_TIMEOUT=5
GRUB_DISTRIBUTOR="$(sed 's, release .*$,,g' /etc/system-release)"
GRUB_DEFAULT=saved
GRUB_DISABLE_SUBMENU=true
GRUB_TERMINAL_OUTPUT="console"
GRUB_CMDLINE_LINUX="resume=/dev/mapper/kh-swap
rd.lvm.lv=kh/root rd.lvm.lv=kh/swap cgroup_disable=files apparmor=0
crashkernel=1024M,high smmu.bypassdev=0x1000:0x17
smmu.bypassdev=0x1000:0x15 rhgb quiet cgroup1_writeback"
GRUB_DISABLE_RECOVERY="true"
```

执行 grub2-mkconfig -o /boot/grub2/grub.cfg, 并重启系统。开机以后通过 cat /proc/cmdline 查看是否有 cgroup1_writeback 字样，若有则开启成功。

确保开启内核特性后，修改 rubik-daemonset.yaml 文件并重新部署 rubik，部分字段如下所示：

```
"agent": {
  "enabledFeatures": [
    "ioCost"
  ]
}
"ioCost": [{
  "nodeName": "k8s-master",
  "config": [
    {
      "dev": "sdb",
      "enable": true,
      "model": "linear",
      "param": {
        "rbps": 10000000,
        "rseqiops": 10000000,
        "rrandiops": 10000000,
        "wbps": 10000000,
        "wseqiops": 10000000,
        "wrandiops": 10000000
      }
    }
  ]
}]
```

以上配置用于标识支持 iocost 对 IO 权重控制特性配置。其类型为数组，数组中的每一个元素由节点名称 nodeName 和设备参数数组 config 组成。

配置键	类型	描述	可选值
nodeName	string	节点名称	kubernetes 集群中的节点名称
cofnig	数组	单个设备的配置信息	/

单个块设备配置 config 参数：

配置键	类型	描述
dev	string	设备名称，仅支持物理设备
model	string	iocost 模型名，当前仅支持 linear

param	/	设备参数，不同模型有不同参数

模型为 linear 时，param 字段支持如下参数：

配置键	类型	描述
rbps	int64	块设备最大读带宽
rseqiops	int64	块设备最大顺序读 iops
rrandiops	int64	块设备最大随机读 iops
wbps	int64	块设备最大写带宽
wseqiops	int64	块设备最大顺序写 iops
wrandiops	int64	块涉笔最大随机写 iops

rubik 会根据以上配置，同时根据用户配置的在线和离线配置(volcano.sh/preemption) 对相关 pod 进行权重限速。

8.1.4. 网络优先级 Qos 控制

rubik 的 netConfig service 提供网络 QoS 的使能开关以及 waterline 和 bandwidth 设置，在 rubik-2.0-daemonset.yaml 中添加相关配置如下：

```
"agent": {
  "enabledFeatures": [
    "netConfig"
  ]
},
"netConfig": {
      "enable": true,
      "waterline": "20mb",
      "bandwidth": "1mb,2mb"
}
```

该功能需要开启特权模式，因此需要在 rubik 的 yaml 文件中安全相关的配置中增加如下字段:

```
securityContext:
  privileged: true
```

执行 kubectl apply -f rubik-daemonset.yaml 部署 rubik。 在业务 pod 中配置相关的优先级，增加相关的配置：

```
annotations:
  volcano.sh/preemptable: "never"
```

此时 rubik 会根据配置的水位线和带宽配置去限制离线业务的网络带宽，当在线业务不超过 waterline 时，离线业务带宽低于 bandwidth 最大值，当在线业务超过 waterline 时，离线业务带宽低于 bandwidth 最小值。

8.1.5. 压力感知驱逐

rubik 提供了实时动态感知当前 cpu，内存和 io 压力并驱逐高负载离线 pod 的能力。rubik 以 some avg10 作为指标。它表示任一任务在 10s 内的平均阻塞时间占比。用户可按需选择对 CPU、内存、IO 资源进行监测，并设置相应阈值。若阻塞占比超过该阈值，则 rubik 按照一定策略驱逐离线 Pod，释放相应资源。若在线 Pod 的 CPU 和内存利用率偏高，rubik 会驱逐当前占用 CPU 资源/内存资源最多的离线业务。若离线业务 I/O 高，则会选择驱逐 CPU 资源占用最多的离线业务。在离线业务由 volcano.sh/preemption 注解标识。

rubik 依赖于 cgroup v1 下的 psi 特性，通过如下方法查看当前内核是否开启 cgroup v1 的 psi 接口：

```
cat /proc/cmdline | grep "psi=1 psi_v1=1"
```

若无相关字段，则需要在启动项中增加"psi=1 psi_v1=1"相关配置并重启。修改 /etc/default/grub 文件内容如下，在 GRUB_CMDLINE_LINUX 最后增加

```
GRUB_TIMEOUT=5
GRUB_DISTRIBUTOR="$(sed 's, release .*$,,g' /etc/system-release)"
GRUB_DEFAULT=saved
GRUB_DISABLE_SUBMENU=true
GRUB_TERMINAL_OUTPUT="console"
GRUB_CMDLINE_LINUX="resume=/dev/mapper/kh-swap rd.lvm.lv=kh/root
rd.lvm.lv=kh/swap cgroup_disable=files apparmor=0 crashkernel=1024M,high
smmu.bypassdev=0x1000:0x17 smmu.bypassdev=0x1000:0x15 rhgb quiet psi=1 psi_v1=1"
GRUB_DISABLE_RECOVERY="true"
```

执行 grub2-mkconfig -o /boot/grub2/grub.cfg, 并重启系统。开机以后通过 cat /proc/cmdline 查看是否有 psi=1 psi_v1=1 字样，若有则开启成功。

使用 psi 功能，rubik 中相关配置修改如下：

```
"agent": {
  "enabledFeatures": [
    "psi"
  ]
}
"psi": {
    "interval": 10,
    "resource": [
      "cpu",
      "memory",
      "io"
    ],
    "avg10Threshold": 5.0
}
```

psi 字段用于标识基于 psi 指标的干扰检测特性配置。目前，psi 特性支持监测 CPU、内存和 I/O 资源，用户可以按需配置该字段，单独或组合监测资源的 PSI 取值。

配置键[=默认值]	类型	描述	可选值
interval=10	int	psi 指标监测间隔（单位：秒）	[10,30]
resource=[]	string 数组	资源类型，声明何种资源需要被访问	cpu, memory, io
avg10Threshold=5.0	float	psi some 类型资源平均 10s 内的压制百分比阈值（单位：%），超过该阈值则驱逐离线业务	[5.0,100]

配置好之后部署 rubik 即可，rubik 会根据当前部署业务的在离线信息进行对应的监控和驱逐。

8.1.6. 潮汐亲和性控制

在 kubernetes 中，创建 Guarenteed 类型的 Pod 有如下要求：

- Pod 中的每个容器都必须指定内存限制和内存请求。
- 对于 Pod 中的每个容器，内存限制必须等于内存请求。
- Pod 中的每个容器都必须指定 CPU 限制和 CPU 请求。
- 对于 Pod 中的每个容器，CPU 限制必须等于 CPU 请求。

对于指定了至少一个 Container 具有内存或 CPU 的请求或限制的 Pod，但不满足 Guarenteed 类型的 Pod，则为 Burstable 类型。

该特性是针对 kubernetes 中的 burstable 类型的 Pod 开发的，针对突发场景下的 pod 并无很好的 cpu 隔离性，导致业务之间干扰严重的问题。使用该特性可以使 pod 在低负载下使用更少的 cpu 核，减少 cpu 之间调度切换的影响，高负载下可以抢占其他 cpu 核，保障业务的响应。

使用该特性需要在 rubik-daemonset.yaml 中启用相关服务并重新部署 rubik，配置如下:

```
        "enabledFeatures": [
          "dynAffinity"
        ]
```

在业务 pod 的 yaml 文件中，部署 burstable 类型的 pod，配置样例如下

```
apiVersion: v1
kind: Pod
metadata:
  name: "burstable-test"
spec:
  containers:
    - image: "stress:test"
      imagePullPolicy: IfNotPresent
      name: burstable-test
      resources:
        limits:
          cpu: 2
        requests:
          cpu: 1
```

使用 kubectl 部署一个 burstable 类型的 pod，此时 rubik 会根据 pod 配置的 cpu requests 资源，执行潮汐亲和性绑定，在 cpuset 的 cgroup 控制器路径下，将绑定的 cpu 写入 pod 对应的目录中的 cpuset.prefered_cpus 文件，从而实现对应的控制功能。

8.1.7. quota burst cpu 柔性限流

该特性允许容器的 cpu 使用率低于 quota 时累积 cpu 资源，并在 cpu 利用率超过 quota 时，使用容器累积的 cpu 资源。Pod 的 quota burst 的配置以 volcano.sh/quota-burst-time 注解的形式，在 Pod 创建的时候配置，或者在 Pod 运行期间通过 kubectl annotate 进行动态的修改，支持离线和在线 Pod。使用该功能首先需要在 rubik-daemonset.yaml 中启用相关服务，配置如下：

```
            "enabledFeatures": [
              "quotaBurst"
            ]
```

然后需要在部署 Pod 时使用如下注解对 pod 进行配置，Pod 的 quota burst 默认单位是 microseconds,

配置样例：

```
        annotations:
          volcano.sh/quota-burst-time : "2000"
```

rubik 会解析该配置，并写入/sys/fs/cgroup/cpu 目录下容器的 cgroup 接口中，如/sys/fs/cgroup/cpu/kubepods/burstable/<PodUID>/<container-longid>/cpu.cfs_burst_us。

8.1.8. 内存分级回收

该功能是基于内核 cgroup 的多级别控制。通过监测节点内存压力，多级别动态调整离线业务的 memory cgroup，尽可能地保障在线业务服务质量。

首先需要在 rubik-daemonset.yaml 中增加相关的配置开关并重新部署 rubik，样例如下

```
    config.json: |
     {
      "agent": {
       "enabledFeatures": [
        "dynMemory"
       ]
      },
     "dynMemory": {
      "policy": "dynlevel"
       }
     }
```

pod 的在线离线信息由 volcano.sh/preemptable 注解指定，rubik 会依据当前节点的内存压力大小，依次调整节点离线应用容器的下列值(/sys/fs/cgroup/memory 目录下容器的 cgroup 中，如 /sys/fs/cgroup/memory/kubepods/burstable/<PodUID>/<container-longid>)：

- memory.soft_limit_in_bytes
- memory.force_empty
- memory.limit_in_bytes
- /proc/sys/vm/drop_caches

8.1.9. 访存带宽和 LLC 限制

rubik 支持业务的 Pod 访存带宽(memory bandwidth)和 LLC(Last Level Cache)限制，通过限制离线业务的访存带宽/LLC 使用，减少其对在线业务的干扰。

前置条件：

- cache/访存限制功能仅支持物理机，不支持虚拟机。
  - x86 物理机，需要 OS 支持且开启 intel RDT 的 CAT 和 MBA 功能，内核启动项 cmdline 需要添加 rdt=l3cat,mba
  - ARM 物理机，需要 OS 支持且开启 mpam 功能，内核启动项需要添加

mpam=acpi。

● 挂载目录/sys/fs/resctrl。 rubik 需要读取和设置/sys/fs/resctrl 目录下的文件，该目录需在 rubik 启动前挂载，且需保障在 rubik 运行过程中不被卸载。

● 权限：SYS_ADMIN. 设置主机/sys/fs/resctrl 目录下的文件需要 rubik 容器被赋有 SYS_ADMIN 权限。

● namepsace: pid namespace. rubik 需要获取业务容器进程在主机上的 pid，所以 rubik 容器需与主机共享 pid namespace。

使用该功能首先需要挂载/sys/fs/resctrl 目录，执行下面命令

```
mount -t resctrl resctrl /sys/fs/resctrl/
```

需要在 rubik-daemonset.yaml 中增加相关的配置开关并重新部署 rubik，样例如下

```
{
  "agent": {
    "enabledFeatures": [
      "dynCache"
    ]
  },
  "dynCache": {
      "defaultLimitMode": "static",
      "adjustInterval": 1000,
      "perfDuration": 1000,
      "l3Percent":{
          "low": 10,
          "mid": 10,
          "high": 100
      },
      "memBandPercent":{
        "low": 10,
        "mid": 10,
        "high": 50
        }
      }
}
```

上述配置配置了 low,mid,high 三个控制组的 LLC 限制百分比和访存带宽百分比，rubik 会在 resctrl 目录（默认为 /sys/fs/resctrl ）下创建 5 个控制组，分别为 rubik_max、rubik_high、rubik_middle、rubik_low、rubik_dynamic。rubik 启动后，将水位线写入对应控制组的 schemata。其中，low、middle、high 的水位线可在 cacheConfig 中配置；max 控制组为默认最大值，dynamic 控制组初始水位线和 low 控制组一致。

离线业务 Pod 启动时通过注解 volcano.sh/cache-limit 设置其 cache level，并被加入到指定的控制组中，如下列配置的 Pod 将被加入 rubik_low 控制组：

```
    annotations:
      volcano.sh/cache-limit: "low"
```

8.1.10. 容器场景内存超分控制

rubik 支持内存超分比，通过主动回收特性将离线任务的内存进行回收，并不断观测 pod 的 psi 指标，不断调整 psi 回收阈值，优先回收压力低于 psi 回收阈值的 pod。

前置条件：

1. 创建离线任务专用 swap 设备，以创建 1G 的块设备作为 swap 设备为例：

```
dd if=/dev/zero of=/var/swapfile bs=1M count=1000
mkswap /var/swapfile
swapon /var/swapfile
```

2. 打开 swap 配置开关

```
echo 2 > /proc/sys/vm/memcg_swap_qos_enable
```

3. 打开 psi 功能

rubik 依赖于 cgroup v1 下的 psi 特性，通过如下方法查看当前内核是否开启 cgroup v1 的 psi 接口：

```
cat /proc/cmdline | grep "psi=1 psi_v1=1"
```

若无相关字段，则需要在启动项中增加"psi=1 psi_v1=1"相关配置并重启。修改 /etc/default/grub 文件内容如下，在 GRUB_CMDLINE_LINUX 最后增加

```
GRUB_TIMEOUT=5
GRUB_DISTRIBUTOR="$(sed 's, release .*$,,g' /etc/system-release)"
GRUB_DEFAULT=saved
GRUB_DISABLE_SUBMENU=true
GRUB_TERMINAL_OUTPUT="console"
GRUB_CMDLINE_LINUX="resume=/dev/mapper/kh-swap
rd.lvm.lv=kh/root rd.lvm.lv=kh/swap cgroup_disable=files apparmor=0
crashkernel=1024M,high smmu.bypassdev=0x1000:0x17
smmu.bypassdev=0x1000:0x15 rhgb quiet psi=1 psi_v1=1"
GRUB_DISABLE_RECOVERY="true"
```

执行 grub2-mkconfig -o /boot/grub2/grub.cfg, 并重启系统。开机以后通过 cat /proc/cmdline 查看是否有 psi=1 psi_v1=1 字样，若有则开启成功。

配置：

需要在 rubik-daemonset.yaml 中增加相关的配置开关并重新部署 rubik，样例如下

```
"agent": {
  ...
"enabledFeatures": [
  "memSave"
  ]
},
"memSave": {
"swapfile": "/var/swapfile",
  "memoryOverloadRatio": 1.5
```

业务 pod 通过以下注解来配置优先级，true 表示离线任务，false 表示在线任务

```
annotations:
  volcano.sh/preemptable: "true"
```

memsave 服务主要流程如下：

1. 启动 rubik，memsave 服务启动
2. 将离线设备的 swapfile 设置为用户配置的 swap 设备
3. 检查离线 pod 是否到达目标超分比，达到则不用回收
4. 如果未达到，则更新 psi 阈值，（初始化 psi 阈值为 0）
5. 收集所有离线 pod 的 psi，排序筛选出 psi 低于 psi 阈值的 pod
6. 回收每个 pod 的 usage * n%
7. 10s 后重复此操作

通过开启 memsave 服务，可以实现对离线业务的内存压缩以达到超分比，提高内存资源利用率的同时减少回收对离线业务的性能影响。

8.1.11. 基于 PSI 的多容器间资源竞争监控指标采集

rubik 支持采集 pod 的 psi 指标，通过 prometheus exporter 的形式进行观测。

前置条件：

1. 内核打开 psi 功能

rubik 依赖于 cgroup v1 下的 psi 特性，通过如下方法查看当前内核是否开启 cgroup v1 的 psi 接口：

```
cat /proc/cmdline | grep "psi=1 psi_v1=1"
```

若无相关字段，则需要在启动项中增加"psi=1 psi_v1=1"相关配置并重启。修改 /etc/default/grub 文件内容如下，在 GRUB_CMDLINE_LINUX 最后增加

```
GRUB_TIMEOUT=5
GRUB_DISTRIBUTOR="$(sed 's, release .*$,,g' /etc/system-release)"
GRUB_DEFAULT=saved
GRUB_DISABLE_SUBMENU=true
GRUB_TERMINAL_OUTPUT="console"
GRUB_CMDLINE_LINUX="resume=/dev/mapper/kh-swap
rd.lvm.lv=kh/root rd.lvm.lv=kh/swap cgroup_disable=files apparmor=0
crashkernel=1024M,high smmu.bypassdev=0x1000:0x17
smmu.bypassdev=0x1000:0x15 rhgb quiet psi=1 psi_v1=1"
GRUB_DISABLE_RECOVERY="true"
```

执行 grub2-mkconfig -o /boot/grub2/grub.cfg, 并重启系统。开机以后通过 cat /proc/cmdline 查看是否有 psi=1 psi_v1=1 字样，若有则开启成功。

配置

需要在 rubik-daemonset.yaml 中增加相关的配置开关并重新部署 rubik，样例如下

```
  "agent": {
    ...
"enabledFeatures": [
    "prometheusExporter"
    ]
  },
  "prometheusExporter": {
"port": "8090",
    "exporteInterval": 10
```

prometheusexporter 服务主要流程如下：

1. 用户配置 rubik 配置文件，enabledFeatures 中写入 prometheusExporter,启动 rubik 服务，prometheusExporter 功能即可开启。
2. 加载用户指定配置
3. 创建 exporter
4. 收集 psi 指标数据，上传给 exporter。

采集指标如下：

指标名称	指标含义	统计指标	含义
cgroup_memory_reclaim	内存回收	avg10 avg60 avg300 total	过去 10s,60s,300s 处于内存回收压力状态，total 为自进程启动以来处于内存回收压力状态的时间，单位为微秒
global_memory_reclaim	全局内存回收	avg10 avg60 avg300 total	过去 10s,60s,300s 处于内存全局回收压力状态，total 为自进程启动以来处于全局内存回收压力状态的时间，单位为微秒
compact	内存压缩	avg10 avg60 avg300 total	过去 10s,60s,300s 处于内存压缩压力状态，total 为自进程启动以来处于内存压缩压力状态的时间，单位为微秒
cgroup_async_memory_reclaim	内存异步回收	avg10 avg60 avg300 total	过去 10s,60s,300s 处于内存异步回收压力状态，total 为自进程启动以来处于内存异步回收压力状态的时间，单位为微秒
swap	交换（swap）	avg10 avg60 avg300	过去 10s,60s,300s 处于交换设备压力状态，total 为自进程启

		total	动以来处于交换设备压力状态的时间，单位为微秒
cpu_cfs_bandwidth	cpu 完全公平调度	avg10 avg60 avg300 total	过去 10s,60s,300s 处于 CPU CFS 带宽压力状态，total 为自进程启动以来处于 CPU CFS 带宽压力状态的时间，单位为微秒
cpu_qos	cpu qos	avg10 avg60 avg300 total	过去 10s,60s,300s 处于 cpu 服务质量压力状态，total 为自进程启动以来处于 cpu 服务质量压力状态的时间，单位为微秒

通过开启 prometheusexporter 服务，可以定时检测 pod 的 psi 压力，直观的体现 pod 的压力大小，方便用户及时响应对 pod 的维护操作。

8.1.12. 在离线混部场景支持 intel 动态频率控制技术

该特性需要整机支持 Intel speed select 技术相关特性，且需要在 BIOS 中打开对应开关，且仅支持 3 代以上 Intel CPU。由于不同整机 BIOS 固件不同，需要咨询对应厂商进行开启对应开关。该特性是针对低负载 cpu 工作在较高频率产生额外能耗的问题而开发。开启时可以使 k8s 在低负载时使用更少的 cpu 频率，从而降低整机功耗；而在高负载时提高 cpu 频率，保证高优先级负载拥有足够的计算资源。

在 rubik-daemonset.yaml 中启用相关服务并重新部署 rubik，配置如下：

```
            "enabledFeatures": [
             "powerManage"  # 添加该字段为开启 power manage 功能；关闭 power manage 功能将“powerManage”删除即可
            ]
          },
          "powerManage": {
            "highFreqRange": [2800,3400],  # 此处[]数字可以修改为适当的数值，也可空着不写，即“[]”，若空着，则 rubik 会默认使用环境中的全部 cpu turbo 频率进行设置，即“[turboFreq, turboFreq]”。
            "lowFreqRange": [0,2400]    # 同上，空着时默认使用环境基频进行设置，即“[0, baseFreq]”。注：baseFreq 和 turboFreq 可以通过“intel-speed-select perf-profile info -l 0”命令查看
          }
        }
```

其中，highFreqRange 和 lowFreqRange 分别表示高频 cpu 和低频 cpu 的工作频率区间，由用户自定义，也可由 rubik 设置默认值。

此外，还需要使用特权模式启动 rubik，并额外挂载部分目录，在 rubik-daemonset.yaml 中的配置如下：

```
      securityContext:
        privileged: true  # 新增该行使 rubik 容器获得相应权限
        ......
      volumeMounts:
      ......
      - name: cpu
        mountPath: /sys/devices/cpu
        readOnly: false
      - name: proc
        mountPath: /proc
        readOnly: false
      - name: kubernetespolicy
        mountPath: /var/lib/kubelet
      - name: isstools
        mountPath: /usr/bin/intel-speed-select
        ......
    volumes:
    ......
    - name: cpu
      hostPath:
        path: /sys/devices/cpu
    - name: proc
      hostPath:
        path: /proc
    - name: kubernetespolicy
      hostPath:
        path: /var/lib/kubelet
    - name: isstools
      hostPath:
        path: /usr/bin/intel-speed-select
```

在业务 pod 的 yaml 中，部署在离线 pod 样例如下：

```
    apiVersion: v1
    kind: Pod
    metadata:
      name: pod-name
      namespace: pod-namespace
      annotations:
        volcano.sh/preemptable: "false"  # “false”表示该 pod 为高优先级，“true”
表示低优先级
    spec:
      containers:
      - name: container-name
        image: workload:test
        imagePullPolicy: IfNotPresent
        resources:
          limits:
            cpu: 3
            memory: "1Gi"
          requests:
            cpu: 3
            memory: "1Gi"
```

该特性根据 k8s cpu manager policy 不同而效果不同。当 policy 为 static 时，该特性仅会令高优先级 guaranteed pod 工作在高频 CPU 上；当 policy 为 none 时，该特性会令所有高优先级 pod 工作在高频 cpu 上，并且会根据实际负载情况动态扩缩高频 cpu 范围。

8.1.13. rubik 容器镜像制作方法

需要在 hostos 环境中构建镜像。

在/opt 目录下新建目录 rubik，安装完 rubik 后，将/var/lib/rubik/目录下的 Dockerfile 文件和 build 目录下全部内容（包括目录本身）拷贝至/opt/rubik 目录中。在/opt/rubik 目录中新建镜像构建脚本文件 build-img.sh，文件内容如后文所示。最后的目录结构应该如下所示：

/opt/rubik

|--build

| |--rubik

|--Dockerfile

|--build-img.sh

在/opt/rubik 目录中执行命令，生成 rubik 服务使用的镜像

bash build-img.sh rubik:2.0.0-1（命令最后是生成的 rubik 镜像名，在后续 rubik 的 yaml 配置文件中使用）

build-img.sh 文件内容如下：

```
-----------------------------------------------（分隔符）
set -ex
tag=$1

yum_config=/etc/yum.conf
target=$(pwd)/baseimage

# option defaults
if [ -f /etc/dnf/dnf.conf ] && command -v dnf &> /dev/null; then
        yum_config=/etc/dnf/dnf.conf
        alias yum=dnf
fi

# clear workspace
rm -rf $target

# install rubik require
yum -c "$yum_config" --installroot="$target" --releasever=/
--setopt=tsflags=nodocs \
                --setopt=group_package_types=mandatory -y install oncn-bwm

# clean yum cache
yum -c "$yum_config" --installroot="$target" -y clean all

# save version
docker build --tag $tag .

# clean workspace
rm -rf "$target"
-----------------------------------------------（分隔符）
```

Dockerfile 内容如下：

```
-----------------------------------------------（分隔符）
```

```
FROM scratch
ADD baseimage /
COPY ./build/rubik /rubik
CMD ["/rubik"]
----------------------------------------------（分隔符）
```

8.1.14. 块 IO 的 IOPS 限流功能使用

8.1.14.1. 概述

Linux Cgroup（Control Groups）是 Linux 内核提供的用于限制、记录、隔离进程组可以使用的资源（cpu、memory、IO 等）的一种机制。Cgroup 里每个子系统（SubSystem）对应一种资源，Cgroup blkio 子系统用于限制块设备 I/O 速率。

8.1.14.2. IOPS 限流功能使用

在 cgroup 子系统目录下通过 mkdir 创建资源组，如在/sys/fs/cgroup/blkio/目录下创建 test 资源组。

```
mkdir /sys/fs/cgroup/blkio/test
```

对指定的设备进行 IOPS 策略限制，如对 major:minor 为 8:32 的设备，限制 IOPS 上限为 16。

```
echo "8:32 16" > /sys/fs/cgroup/blkio/test/blkio.throttle.write_iops_device
```

8.2. 虚机高低优先级混部

8.2.1. 概述

虚拟机混合部署是指把对 CPU、IO、Memory 等资源有不同需求的虚拟机通过调度方式部署、迁移到同一个计算节点上，从而使得节点的资源得到充分利用。

虚拟机混合部署的场景有多种，而虚拟机高低优先级调度也是其中的一种实现方法。在单机的资源调度分配上，区分出高低优先级，即高优先级虚机和低优先级虚机发生资源竞争时，资源优先分配给前者，严格保障其 QoS。

8.2.2. 实现方案

用户创建 flavor 或创建虚机时，可指定其优先级属性。但优先级属性不影响 Nova 现有的资源模型及节点调度策略，即 Nova 仍按正常流程选取计算节点及创建虚机。

虚机高低优先级特性主要影响虚机创建后单机层面的资源调度分配策略。高优先级虚机和低优先级虚机发生资源竞争时，资源优先分配给前者，严格保障其 QoS。

为了让高优先级虚拟机达到最佳性能，推荐高优先级虚拟机 vCPU 与物理 CPU 一对一绑核。为了让低优先级虚拟机充分利用空闲物理资源，推荐低优先级虚拟机 vCPU 范围绑核，且绑核范围覆盖高优先级虚拟机绑核范围。

同时为了防止出现因高优先级虚拟机长时间占满 CPU 导致低优先级虚拟机无法被调度的情况，需要预留少量低优先级虚拟机专用的 CPU，该部分 CPU 不可让高优先级虚拟机绑定，且要求让低优先级虚拟机绑定。

在 OpenStack Nova 中引入虚拟机高低优先级技术，再配合 Skylark QoS 服务能力，可以一定程度上满足虚拟机的混合部署要求。

8.2.3. 使用方式

[前置要求]

1. 存在已安装部署 openstack T 版的服务器，openstack T 版服务部署可以参考适配文档；

2. 服务器操作系统为 Kylin HostOS V10，并且已安装 skylark 服务，且内核版本符合 skylark 要求；

[配置修改]

打开/etc/nova/nova.conf，修改[compute]内的 cpu_dedicated_set、cpu_shared_set 和 cpu_priority_mix_enable，其中 cpu_dedicated_set 表示高优先级虚机使用的 CPU 核心，cpu_shared_set 指低优先级虚机使用的 CPU 核心，cpu_priority_mix_enable 表示是否允许低优先虚机同时使用 cpu_dedicated_set 的 CPU 核心，默认值是 False。

在此以物理机存在 0-11，共 12 个 CPU 核数为例：

```
[compute]
cpu_dedicated_set=0-7
cpu_shared_set=8-11
cpu_priority_mix_enable=True
```

重启 openstack-nova-compute 服务

```
systemctl restart openstack-nova-compute
```

[使用步骤]

在执行 openstack 命令行命令前，先导入环境变量，一般在 openstack 部署阶段会生成：

```
source ~/admin-rc
```

1. 创建 openstack flavor 实例类型，分别设置为高优先级虚拟机与低优先级虚拟机模型，示例如下：

```
openstackflavorcreate--ram8192--disk50--vcpus8--public--propertyhw:cpu_priority='low'low_prio
openstackflavorcreate--ram8192--disk50--vcpus8--public--propertyhw:cpu_priority='high'--propertyhw:cpu_policy='dedicated'high_prio
```

2. 参考《银河麒麟操作系统 Kylin HostOS V10 Openstack T 版单节点适配指南》，使用刚刚创建的实例类型创建实例。

创建实例后，可以在实例所在的计算节点上执行 virsh list 查看虚拟机 id，并且通过使用 virsh vcpuinfo 命令查看虚拟机的 vcpu 绑定情况。

以上面的配置为例，可以发现高优先级的虚拟机的 vcpu 与主机上 0-7 号 CPU 绑定，并且是独占式；低优先级的虚拟机的 vcpu 与主机上 8-11 号 CPU 中绑定，并且是非独占式绑定，可以有多个 vcpu 绑定到同一 CPU 上。

9. 云场景调优

9.1. openstack S2500

openstack 是当前云场景下常用的编排平台，在默认情况下其提供的性能足够大多数场景使用，但一些情况下需要极致性能时，需要根据 CPU 拓扑对计算负载进行绑核。

openstack 默认的绑核策略针对的是通用架构,并未对飞腾 CPU S2500 进行特殊优化，因此针对该情况，做了深度修改，使得在云场景下 kubernetes 和 openstack 在飞腾 CPU S2500 上的计算负载性能达到最优，使用方式如下：

1. 安装最新版的 openstack-nova-23.2.2-1.p02.ky10 版本 ，该版本默认集成了飞腾 S2500 的资源调配逻辑

2. 在控制节点主机执行以下命令，创建实例类型 flavor

```
openstack flavor create <FLAVOR-NAME> --vcpus=<CPU 数 量 >
--ram=<内存大小 MB> --disk=<磁盘大小 GB>
```

以 4C8G-50G 规格为例：

```
openstack flavor create test-flavor --vcpus=4 --ram=8192 --disk=50
```

3. 设置 cpu 绑定策略 cpu_policy 为独占 pcpu，cpu_thread_policy 为 isolate:

```
openstack flavor set 'flavorid' --property hw:cpu_policy=dedicated \
--property hw:cpu_thread_policy=isolate
```

4.给 flavor 指定属性，分散 numa，对 cpu 数量和内存大小在每个 numa 上进行分配：

```
openstack flavor set FLAVOR-NAME \
--property hw:numa_nodes=FLAVOR-NODES \
--property hw:numa_cpus.N= FLAVOR-CORES\
--property hw:numa_mem.N=FLAVOR-MEMORY
```

参数说明：

- FLAVOR-NODES: (整数) 限制虚拟机 vCPU 线程运行的可选 NUMA 节点数量。如果不指定，则 vCPU 线程可以运行在任意可用的 NUMA 节点上。
- N: (整数) 应用 CPU 或内存配置的虚拟机 NUMA 节点，值的范围从 0 到 FLAVOR-NODES - 1。比如为 0, 则运行在 NUMA 节点 0; 为 1, 则运行在 NUMA 节点 1.
- FLAVOR-CORES: (英文逗号分隔的整数) 映射到虚拟机 NUMA 节点 N 上的虚拟机 vCPU 列表。如果不指定，vCPU 在可用的 NUMA 节点之间平均分配。
- FLAVOR-MEMORY: (整数，单位 MB) 映射到虚拟机 NUMA 节点 N 上的虚拟机内存大小。如果不指定，则内存平均分配到可用 NUMA 节点。

5. 在实际分散 numa 前，请执行以下命令确认主机上可用的 numa 节点数量以及内存条数量：

```
numactl -H
```

如果设置的 hw:numa_mem.N 条数超过主机上可用的内存条数，则该实例类型无法正常创建实例。

6.以 4C8G 格式的虚拟机为例，确保主机上 numa 节点数与内存条数满足其要求，创建实例类型如下：

```
openstack flavor set <flavorid> \
--property hw:numa_nodes=4 \
--property hw:numa_cpus.0=0 \
--property hw:numa_cpus.1=1 \
--property hw:numa_cpus.2=2 \
--property hw:numa_cpus.3=3 \
--property hw:numa_mem.0=2048 \
--property hw:numa_mem.1=2048 \
--property hw:numa_mem.2=2048 \
--property hw:numa_mem.3=2048
```

7. 使用新建的实例类型创建实例，openstack 会自动检查计算节点的 CPU model，如果是飞腾 S2500，则会使用针对飞腾 S2500 的绑核调度策略。

openstack 会将 4 个 cpu 分布到 4 个 NUMA 节点上，且每一个 cpu 独占一个 L2 缓存，该策略在飞腾 S2500 机器上最大限度的保证了计算负载的性能和稳定性。

9.2. kubernetes S2500

kubernetes 是当前云场景下最常用的容器编排平台，在默认情况下两者提供的性能足够大多数场景使用，但一些情况下需要极致性能时，需要根据 CPU 拓扑对计算负载进行绑核。kubernetes 默认的绑核策略针对的是通用架构，并未对飞腾 CPU S2500 进行特殊优化，因此针对该情况，做了深度修改，使得在云场景下 kubernetes 在飞腾 CPU S2500 上的计算负载性能达到最优，使用方式如下

1. 安装 kubernetes，该版本默认集成了飞腾 S2500 的资源调配逻辑

2. 配置 kubelet 服务，在启动命令中增加如下参数

```
    --feature-gates=CPUManagerPolicyAlphaOptions=true,CPUManagerPoli
cyOptions=true
    --cpu-manager-policy-options=distribute-cpus-across-numa=true
    --cpu-manager-policy=static
    --kube-reserved=cpu=<自己定义需要保留的 cpu 个数>
    --system-reserved=cpu=<自己定义需要保留的 cpu 个数>
```

新增的启动参数说明：

--feature-gates 指定当前版本要使用 CPUManagerPolicyAlphaOptions，仅当该参数指定时；--cpu-manager-policy-options=distribute-cpus-across-numa=true 起作用；

--cpu-manager-policy-options 指定在特定 NUMA 节点集中分配 CPU 时的分配策略为平均分配到多个 NUMA 节点上；

--cpu-manager-policy 指定 cpu-manager 分配 cpu 时采用静态分配的策略而非 none；

--kube-reserved 和 --system-reserved 参数指定系统和 kubelet 要保留的 CPU 个数，不指定的话 cpu-manager 无法正常使用。

删除/var/lib/kubelet/cpu_manager_state, 并重启 kubelet。

1.部署 Guarenteed 类型的 Pod，即同时配置 CPU 和内存使用限制，且 requests 与 limits 数值相等，举例如下：

```
    apiVersion: apps/v1
    kind: Deployment
    metadata:
      name: stream-numa-test
      namespace: default
      labels:
        app: stream-numa-test
    spec:
      replicas: 1
      selector:
        matchLabels:
          app: stream-numa-test
      template:
        metadata:
          labels:
            app: stream-numa-test
        spec:
          containers:
            - image: 'stream:latest'
              imagePullPolicy: IfNotPresent
              name: stream-numa-test
              args: [ "bash", "-c", "sleep 10000" ]
```

```
        resources:
          limits:
            cpu: "8"
            memory: "8G"
          requests:
            cpu: "8"
            memory: "8G"
      restartPolicy: Always
```

该配置指定了一个 Guarenteed 类型的 Pod，对于该类型的 Pod，kubelet 会将所需的 CPU 个数按飞腾 S2500 的资源调配逻辑进行分配，对于上面的配置，kubelet 会将 8 个 CPU 优先分配到 L2 缓存未被占用的 CPU 上，从而减少和其他 Pod 的 L2 缓存抢占，该策略在飞腾 S2500 机器上最大限度的保证了计算负载的性能和稳定性。

9.3. ceph 存储节点优化

Ceph 是一个专注于分布式的、弹性可扩展的、高可靠的、性能优异的存储系统平台((由 C++编写完成，提供软件定义、统一存储解决方案)，可以同时支持块设备、文件系统和对象网关三种类型的存储接口。

在 Hostos 底座中，增加了 Tuned 优化 ceph 性能的支持，参照官方文档部署 ceph 后，在系统中可以观测到 ceph-osd 进程，执行如下命令来优化 ceph 性能；

1） 验证安装包是否含有 ceph 配置文件：

```
tuned-adm list | grep ceph
```

2） 验证新增的 ceph 配置项

```
rpm -ql tuned |grep ceph 安装后查看已存在文件
```

3）配置 ceph 优化选项

```
tuned-adm profile ceph-performance
```

4）验证配置是否成功

```
tuned-adm active
```

10. 系统安全

10.1. 安全套件 SDK

10.1.1. 简介

为第三方安全套件提供安全功能接口，也就是安全套件 SDK。

10.1.2. 主要功能

为安全套件提供以下能力的接口：

1）系统安全功能：包括进程防杀死、文件防篡改、内核模块防卸载、软件防卸载、进程管控。

2） IO 管控：网卡管控、光驱管控、打印管控、扫描管控、存储设备管控、USB 等接口管控。

3） 主机审计：打印日志、刻录日志等。

4）安全登录：壁纸设置等。

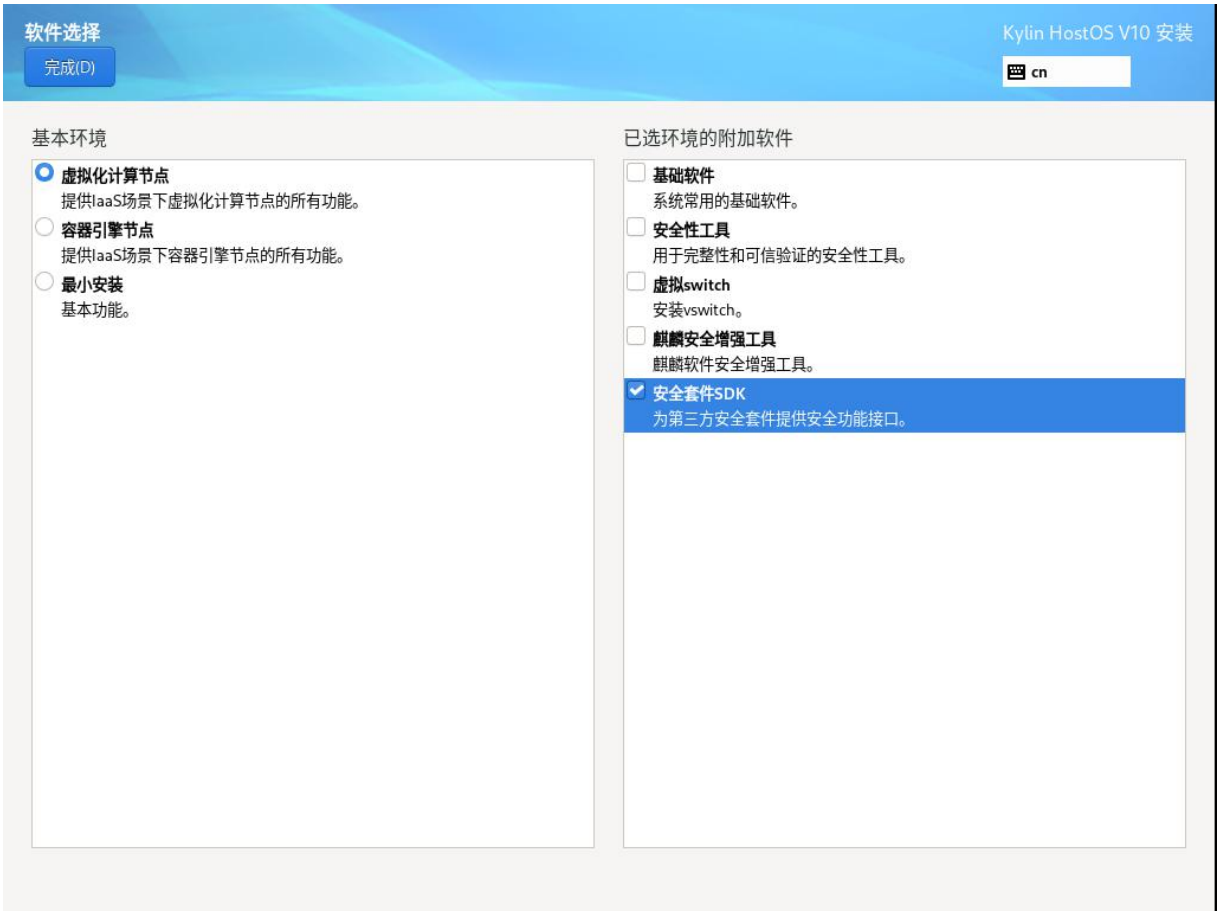
10.1.3. 安装使用

10.1.3.1. 安装安全套件 SDK

进入安装界面，点击“软件选择”，右侧勾选“安全套件 SDK”。

其他选项根据需要勾选。

如图所示，继续安装直至系统安装完毕。



10.1.4. 安装第三方安全套件软件包

系统安装完毕后，按第三方安全套件软件包安装说明，安装第三方安全套件软件。

10.1.5. 状态检查

安装套件后，重启操作系统，执行 getstatus 命令。

kysec 开启，fpro、kmod、ppro 开关为 on。

状态如下所示，可以正常使用安全套件。

10.2. 安全增强工具

面对日益复杂的网络环境，确保系统的安全性已成为一项至关重要的任务。本章节将介绍麒麟安全增强工具和技术，旨在为系统管理员提供多层防护策略，构建更为稳固的安全防线。

10.2.1. KYSEC 安全机制

银河麒麟云底座操作系统 V10 结合国产操作系统特点和现有国内外操作系统的安全机制，提出了基于标记的软件执行控制机制，实现对系统应用程序标记识别和执行约束，确保应用来源的可靠性和应用本身的完整性。执行控制机制控制文件执行、模块加载和共享库使用，分为系统文件、第三方应用程序，其中只允许具有合法标记的文件执行，任何网络下载、拷贝等外来软件均被禁止执行。

10.2.1.1. 安全状态设置

KYSEC 共有两种状态；强制模式（enforcing）、软模式（permissive）、关闭模式（disable）。强制模式下不能执行非法应用程序，即禁止用户执行没有合法标记的应用程序，而软模式下，只是记录非法应用程序的执行行为，不禁止用户操作，关闭模式下，不记录非法应用程序的执行行为也不禁止用户操作。系统默认 KYSEC 安全状态为关闭状态，且执行控制、应用联网管控、文件保护、内核保护、进程防杀保护状态都处

于关闭状态，三权分立关闭。

查看 kysec 安全状态：

```
#getstatus
KySec status: enforcing
  exectl：on
  netctl：on
  fpro:on
  kmod:on
  ppro:on
selinux status:disable
apparmor status:disable
box status:disable
```

设置 kysec 安全状态为软模式：

```
#setstatus kysec -s permissive
```

设置 kysec 安全状态为强制模式：

```
# setstatus kysec -s enforcing
```

10.2.1.2. 文件保护功能

文件保护功能开启时，受保护的文件不能被修改、重命名、删除。这样能保护系统重要的配置文件不被误操作修改。

给文件设置文件保护的标记：

```
#kysec_set fpro -v readonly -f /tmp/testfile
```

给文件移除文件保护的标记：

```
#kysec_set fpro -v none -f /tmp/testfile
```

给文件/tmp/testfile 设置文件保护的标记后，则禁止修改,重命名,删除/tmp/testfile 文件。如下操作则会提示失败。

```
#echo “test1” >> /tmp/testfile
#mv /tmp/testfile /tmp/abc
#rm -f /tmp/testfile
```

10.2.1.3. 模块防卸载功能

模块防卸载功能开启时，受保护的模块不能被卸载。

给模块设置保护：

```
#kysec_set kmod -v deny -f /tmp/testmod.ko
```

给模块移除保护：

```
#kysec_set fpro -v allow -f /tmp/testmod.ko
```

模块防卸载功能开启时，受保护的模块不能被卸载。

给模块设置保护：

```
#kysec_set kmod -v deny -f /tmp/testmod.ko
```

给模块移除保护：

```
#kysec_set fpro -v allow -f /tmp/testmod.ko
```

10.2.1.4. 进程防杀功能

进程防杀功能开启时，受保护的进程不能被 kill。

给进程设置保护：

```
#kysec_set ppro -v deny -f /usr/bin/top
```

给进程移除保护：

```
#kysec_set ppro -v allow -f /usr/bin/top
```

10.2.1.5. 应用联网管控功能

应用联网管控功能开启时，设置的应用不能连接互联网。

给应用设置保护：

```
#kysec_set netctl -v deny -f /usr/bin/ssh
```

给应用移除保护：

```
#kysec_set netctl -v allow -f /usr/bin/ssh
```

10.2.1.6. 执行控制

银河麒麟云底座操作系统 V10 添加了对文件执行、模块加载和共享库使用的控制，分为系统文件、第三方应用程序，其中只允许具有合法标记的文件执行，任何网络下载、

拷贝等外来软件均被禁止执行。本来可运行的可执行文件在删除、修改、拷贝、移动、重命名等操作后，会被禁止执行。

（1）可执行文件的执行控制

如下情况的/tmp/ls 则执行失败。

```
#cp /bin/ls /tmp
#/tmp/ls
```

但管理员设置/tmp/ls 的 kysec 标记后，用户则可执行/tmp/ls 程序：

```
# kysec_set exectl -v original -f /tmp/ls
#/tmp/ls
```

篡改/tmp/ls 程序内容后，该程序不能再执行：

```
#echo “ ” >> /tmp/ls
#/tmp/ls
```

（2）可执行脚本的执行控制

创建脚本文件/tmp/test.sh 并添加执行权限,内容如下：

```
#!/bin/bash
echo “test”
#chmod +x /tmp/test.sh
```

但用户使用如下命令，都无法执行程序：

```
#/tmp/test.sh
#bash /tmp/test.sh
```

管理员用户设置该文件的 kysec 标记后，程序就能正常执行。

```
#kysec_set exectl -v original -f /tmp/test.sh
```

（3）内核模块的执行控制

用户将可加载的内核模块复制到主目录,再尝试加载拷贝的内核模块,因为 KYSEC 功能，则加载失败：

```
#cp /lib/modules/`uname -r`/kernel/net/netfilter/nf_conntrack_ftp.ko ~
#insmod ~/nf_conntrack_ftp.ko
```

修改拷贝的内核模块标记，则可以加载成功。

```
#kysec_set exectl -v original -f ~/nf_conntrack_ftp.ko
#insmod ~/nf_conntrack_ftp.ko
```

10.2.2. 强制访问控制

在计算机安全领域指一种由操作系统约束的访问控制，目标是限制主体或发起者访问或对对象或目标执行某种操作的能力。在实践中，主体通常是一个进程或线程，对象可能是文件、目录、TCP/UDP 端口、共享内存段、I/O 设备等。主体和对象各自具有一组安全属性。每当主体尝试访问对象时，都会由操作系统内核强制施行授权规则——检查安全属性并决定是否可进行访问。任何主体对任何客体的任何操作都将根据一组授权规则（也称策略）进行测试，决定操作是否允许。

主体和客体：主体指的是访问者(一般为一个进程)，客体指的是被访问的资源。根据资源的类型不同，在策略配置语言中被分为不同的客体，如 dir、file、process 等。

安全上下文：指标记在所有事物上的扩展属性，包括 SELinux 用户、角色、域或类型，如果使用 MLS(多级安全，以下会有介绍)策略还包括安全级。

类型和域：是分配给一个对象并决定谁可以访问这个对象，域对应进程，类型对应其他对象。

域转换：进程以给定的进程类型运行的能力称为域转换，需满足三个转换条件：一个进程新的域类型有对一个可执行文件类型的 entrypoint 访问权限，进程的当前（或以前）域类型对入口点文件类型拥有 execute 访问权限，进程的当前域类型对新的域类型拥有 transition 访问权限。

角色：和一些域相关联，决定哪些域可以使用。

安全策略用户：和标准 linux 用户对应，影响哪个域可以进入。

10.2.2.1. SELinux 基础策略

银河麒麟云底座操作系统 V10 根据系统安全需求，定制了系统图形登录功能策略、三权分立功能策略、审计服务策略、执行控制功能策略、白名单功能策略、kvm/lxc 等系统功能策略、系统使用修订桌面常用工具策略、系统启动时自动标记脚本功能。根据不同应用场景，定制了 ukmls、ukmcs 和 target 三套 SE 基础策略。

10.2.2.2. 安全策略模式切换

系统在正常启动后就进入了允许模式。允许模式是为了方便我们服务器配置，即使我们的安全策略没有允许这样的操作仍可以执行，但会被审计下来；强制模式是只要没有这样的规则操作就不被允许。我们的安全服务器系统在普通模式下使用的就是允许模式的策略，在默认模式下则使用强制模式的策略来加固我们服务器的安全。

（1）当前安全策略模式查询

我们提供了一些工具命令可以查询当前的安全策略模式，如/usr/sbin/getenforce 和/usr/sbin/sestatus。所有用户都可以通过这些命令进行查询。

getenforce 命令输出若为 Permissive 则表示当前处于允许模式，若为 Enforcing 则表示当前处于强制模式。

sestatus 命令的输出可能为以下内容：

```
#sestatus
SELinux status:                 enabled
SELinuxfs mount:                /sys/fs/selinux
SELinux root directory:         /etc/selinux
Loaded policy name:             ukmcs
Current mode:                   permissive
Mode from config file:          enforcing
Policy MLS status:              enabled
Policy deny_unknown status:     allowed
Memory protection checking:     actual (secure)
Max kernel policy version:      30
```

它不仅反应了当前安全策略的模式，同时也可以得到其他安全策略的信息：安全策略开启状态、安全策略文件系统挂载位置、配置文件中的默认模式、安全策略版本和安全策略目录名称。

（2）安全策略模式修改

为了安全考虑所以系统一启动都会进入到强制模式，但是也提供了可以修改当前安全策略模式的命令。

setenforce 1 设置为强制模式；

setenforce 0 设置为允许模式。

系统三个管理员中只有安全管理员可以使用 setenforce 1 进入到强制模式，也只有

安全管理员可以使用 setenforce 0 进入到允许模式。

（3）禁用安全策略

在系统出现故障时我们也可以先禁用我们的安全策略进入到维护模式进行修复。进入方法如下：

a)系统启动时进入 grub 菜单。

b)输入 grub 密码修改内核启动参数，将参数 selinux=1 enforcing=1 修改为 selinux=1 enforcing=0 。

（4）启动系统。

这样系统就直接进入到维护模式，由 root 用户对系统进行修复。

10.2.2.3. 策略配置命令

为了方便用户对安全策略进行配置，我们也提供了以下命令对当前安全策略进行配置：

10.2.2.3.1. 查看标记命令

系统中一些常用命令的-Z 选项都提供了查看文件或进程安全上下文的功能，如：ls -Z id -Z 和 ps -Z,这些命令加了-Z 选项后就会显示该文件或进程的安全上下文,如 ls -Z 一个文件，输出以下内容：

```
-rw-r--r--. root root sysadm_u:object_r:admin_home_t:s0
```

其中 sysadm_u 就表示该文件属于 sysadm_u 安全策略用户，object_r 表示该文件属于客体类型的角色，admin_home_t 表示该文件的类型，我们的策略就是根据这些标记来制定规则进行访问控制的。

10.2.2.3.2. 修改标记命令

安全管理员可以通过一些命令对系统中所有文件的安全上下文进行操作，如：chcon、setfiles 和 restorecon 命令。

/bin/ls 命令为 bin_t 类型，安全管理员可以输入 chcon -t sbin_t /bin/ls 将 bin_t 修改为 sbin_t 类型。

chcon 命令的-u 参数可以修改该文件安全上下文中的安全策略用户，-r 参数可以修改它的角色，-l 参数可以修改它的安全级别，其他具体使用可以查看它的帮助手册。

setfiles 命令可以根据我们的安全策略的标记文件对系统中所有文件进行标记。如：

```
setfiles /etc/selinux/targeted/contexts/files/file_contexts /
```

restorecon 命令可以恢复该文件的默认安全上下文，如：restorecon -r filename。

在对文件的安全上下文标记进行操作时首先都会检查这个标记的正确性，然后再去进行操作，具体使用可以查看用户手册。

10.2.2.3.3. 用户、角色和域的关系

登录程序如（login，sshd）负责映射 Linux 用户到安全策略用户，在登录时，如果安全策略用户标识符恰好和一个 Linux 用户标识符完全相同，匹配的安全策略用户标识符成为初始 shell 进程安全上下文的用户标识符。安全管理员也可使用 semanage 命令完成映射，如：semanage login -a -s charlie_u charlie，表示 Linux 用户 charlie 作为安全策略用户 charlie_u 登录系统。

使用 semanage login -l 命令可以查看，如下图：

```
#semanage login -l

登录名        SELinux 用户      MLS/MCS 范围    服务
__default__       user_u        s0-s0:c0.c1023      *
auditadm       auditadm_u     s0-s0:c0.c1023      *
root           root           s0-s0:c0.c1023      *
secadm        secadm_u       s0-s0:c0.c1023      *
```

一个用户可以对应多个角色，但在同一时刻只能作为其中一个角色，可以通过策略管理工具 semanage 输入 semanage user -a -R mgr_r -R charlie_r -P user charlie_u 来创建一个可以作为角色 mgr_r 和 charlie_r 的安全策略用户 charlie_u。

可通过 semanage user -l 进行查看，如下图：

```
#semanage user -l
                标记中       MLS/      MLS/
SELinux 用户    前缀    MCS 级别    MCS 范围    SELinux 角色
auditadm_u    audadm     s0      s0-s0:c0.c1023   auditadm_r
guest_u       user       s0      s0               guest_r
root          sysadm     s0      s0-s0:c0.c1023   sysadm_r
```

```
secadm_u    secadm    s0     s0-s0:c0.c1023   secadm_r
```

角色仅仅是一套域类型的集合，方便与用户建立联系，具体的访问控制都是通过进程的类型和客体的类型根据制定的规则来进行控制。

10.2.2.3.4. 端口配置

系统中所有端口同样也被标记了类型，安全管理员可以通过：

```
semanage port -a -t vnc_port_t -p udp 222
```

来添加一个标记了类型的端口。

可通过 semanage port -l 进行查看，如下图：

```
#semanage port -l
SELinux 端口类型                协议      端口号
afs3_callback_port_t        tcp     7001
afs3_callback_port_t        udp     7001
afs_bos_port_t              udp     7007
afs_fs_port_t               tcp     2040
afs_fs_port_t               udp     7000, 7005
afs_ka_port_t               udp     7004
```

10.2.3. 三权分立机制

三权分立系统在满足安全系统标准要求的同时，尽量减少对系统的改动，大大的提高了系统的可用性和引用性。

主要设计思想是，将传统意义上超级管理员 root 的权能进行划分，分别为 root(uid=0)、secadm(uid=600)和 auditadm(uid=700)分别作为超级用户的别名存在，它们分别对应 selinux 三权分立角色中的一个角色，从而保持传统系统用户身份鉴别系统结构的同时，达到三权分立的目的。

三权分立系统是对 linux 现有用户身份认证机制的扩展，增加部分功能主要是为解决传统认证机制无法支持三权分立的用户身份的鉴别。三权系统使用传统 linux 用户身份鉴别机制的认证系统之外可以额外采用双因子验证方式进行身份鉴别，这套系统是专门定制开发，且符合公安部安全操作系统相关标准。

10.2.3.1. 功能

三权系统功能包括两部分：

1、实现对从一个 UID 为 0 的超级用户衍生出的三个管理员用户的身份认证功能，这个认证系统要支持包括登录系统和用户身份切换等所有可能涉及身份权限切换的情况，同时本三权系统对其他非管理员用户是透明的。

2、实现在每次切换管理员身份时，可以按照不同管理员指派不同的 SELinux 角色给用户。

上面两个功能实现的前提是，本三权系统不能影响到系统中传统身份鉴别系统对非管理员用户的所有验证操作，即新三权系统只对三个管理员用户有作用，对任何其他用户是透明的。

10.2.3.2. 示例

只有系统管理员具有“分区管理功能”。该功能使系统中只有系统管理员可对系统磁盘进行分区管理。当三权系统的安全状态为 strict 模式时。系统管理员可在图形界面下，点击应用程序->工具->磁盘。系统管理员可以成功打开分区编辑器。安全管理员、审计管理员的开始菜单中没有分区编辑器的图标显示。

10.2.3.3. 三权用户命令集

三权用户命令集见下表。

用户	命令	备注
系统管理员	reboot	系统重启命令
	init	管理系统运行模式
	ifconfig	网络接口管理（设置类的操作）
	rpm	软件包管理
	useradd	添加用户。如：useradd abc
	passwd	修改普通用户密码。如：passwd abc
	userdel	删除用户。如：userdel -r abc
	insmod	插入模块
	rmmod	删除模块
	iptables	防火墙相关
	system-config-printer	打印机配置

	mount	分区挂载（设置类的操作）
安全管理员	setenforce	开启和关闭 selinux 的强制模式
	checkpolicy	将可读策略文件编译生成二进制策略文件，该文件和所在文件夹必须为 policy_src_t 类型。 如： checkpolicy -M policy.conf -o policy.24
	load_policy	如：load_policy -bq(服务器) 更换内核中的安全策略，保持使用当前的 Boolean 值
	chcon	修改文件和文件夹的安全上下文。如： chcon -t bin_t filename
	chcat	修改文件和文件夹的敏感级。如：chcat s0:c0.c255 filename
	fixfiles	检查修复文件安全上下文。如：fixfiles -F restore filename
	setfiles	初始化文件标记，检查标记正确性。如：setfiles /etc/selinux/targeted/contexts/files/file_conte xts /
	restorecon	恢复默认文件安全上下文 如：restorecon -r filename
	semanage	SELinux 管理工具命令 如： semanage user -l semanage login -l semanage user -m -R secadm_r secadm_u
	semodule	管理 SELinux 策略模块 如： semodule -l semodule -i modulename
	setsebool	修改 SELinux 中的 bool 值来修改策略 如：setsebool xdm_sysadm_login = 1
	audit2allow	显示访问被拒绝后应添加的允许规则 如： audit2allow -a audit2allow -i /tmp/dmesg（将 dmesg 输出到/tmp/dmesg 中）
	audit2why	根据访问被拒绝的审计信息显示被拒绝的原因。 如：audit2why < /tmp/audit.log （将 /var/log/audit/audit.log 拷贝到/tmp 下）
审计管理员	aureport	根据审计信息文件统计登录情况，avc，pid，file，event，config 等众多信息。如： aureport -au aureport -a aureport -p
	ausearch	搜索审计文件中的信息。如： ausearch -gi 0
	auditd	启停审计服务。如： systemctl start/stop/restart auditd
	auditctl	显示服务状态，审计规则相关 如：auditctl -s

10.2.3.4. 启用与关闭

采用如下方式开启三权分立安全功能：

1）执行下述命令开启三权分立安全功能：

```
security-switch --set strict
```

2）重启系统，使得修改生效

3）通过 security-switch --get 命令查看当前修改后的增强安全状态中“三权分立”为启用，命令如下：

```
security-switch --get
```

显示当前安全级别信息：

```
当前安全级别:strict
-----------------------------------------------------------------
 安全模块     |     当前状态     | 默认状态
-----------------------------------------------------------------
SELinux          启用(Enforcing)    启用(Enforcing)
三权分立         启用               启用
Kysec          启用(enforcing)      启用(enforcing)
```

采用切换其他安全模式可以关闭三权分立安全功能：

```
security-switch --set default
```

10.2.4. 动态度量

动态度量组件是一个用于监控软件及系统内核运行状态下是否有被篡改的系统安全插件。该组件实现了针对软件及内核的关键段的周期性度量公共能，当被监控对象关键段(如：只读数据段、代码段)内容发生篡改后，动态度量组件会通过日志系统记录篡改事件，并可以根据策略配置决策是否终止发生篡改的程序。

10.2.4.1. 用法

10.2.4.1.1. 检测环境

检测软件包 lkrg-kyexten、libkydima 是否已安装：

```
# rpm -qa | grep -e kydima -e lkrg
```

```
[root@localhost 4.19.90-89.4.v2401.ky10.x86_64]# rpm -qa | grep -e kydima -e lkrg
lkrg-kyextend-0.9.6-01.se.13.ky10.x86_64
libkydima-1.1.0-07.ky10.x86_64
```

如果未安装则安装软件包：

```
# yum install lkrg-kyextend
```

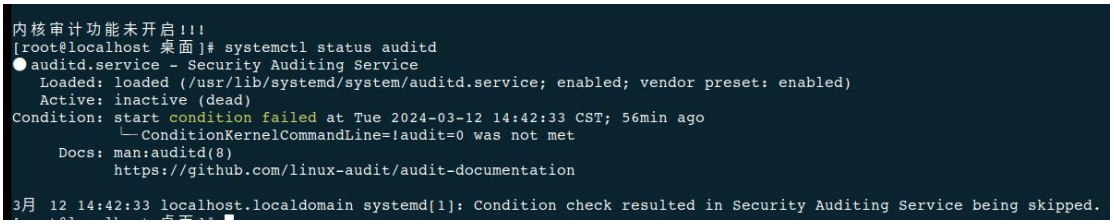
```
[root@localhost 桌面]# yum install lkrg-kyextend
Last metadata expiration check: 0:01:09 ago on 2024年03月12日 星期二 13时49分54秒.
Dependencies resolved.
================================================================================
 Package          Architecture      Version                  Repository     Size
================================================================================
Installing:
 lkrg-kyextend    x86_64            0.9.6-01.se.13.ky10      ks10-adv-os   5.5 M
Installing dependencies:
 dkms             noarch            2.6.1-6.ky10             ks10-adv-os    78 k

Transaction Summary
================================================================================
Install  2 Packages

Total download size: 5.6 M
Installed size: 13 M
Is this ok [y/N]: y
Downloading Packages:
(1/2): dkms-2.6.1-6.ky10.noarch.rpm                 321 kB/s |  78 kB     00:00
(2/2): lkrg-kyextend-0.9.6-01.se.13.ky10.x86_64.rpm 6.5 MB/s | 5.5 MB     00:00
--------------------------------------------------------------------------------
Total                                               6.6 MB/s | 5.6 MB     00:00
Running transaction check
Transaction check succeeded.
Running transaction test
Transaction test succeeded.
Running transaction
  Preparing        :                                                        1/1
  Installing       : dkms-2.6.1-6.ky10.noarch                               1/2
  Running scriptlet: dkms-2.6.1-6.ky10.noarch                               1/2
  Installing       : lkrg-kyextend-0.9.6-01.se.13.ky10.x86_64               2/2
  Running scriptlet: lkrg-kyextend-0.9.6-01.se.13.ky10.x86_64               2/2

Creating symlink /var/lib/dkms/lkrg-kyextend/0.9.6/source ->
                 /usr/src/lkrg-kyextend-0.9.6

DKMS: add completed.

Kernel preparation unnecessary for this kernel.  Skipping...

Building module:
cleaning build area...
'BIZH=/lib/modules/4.19.90-89.1.v2401.ky10.x86_64/build make...
cleaning build area...

DKMS: build completed.

kyrg.ko.xz:
Running module version sanity check.
 - Original module
   - No original module exists within this kernel
 - Installation
   - Installing to /lib/modules/4.19.90-89.1.v2401.ky10.x86_64/updates/
Adding any weak-modules

depmod....

DKMS: install completed.

  Running scriptlet: lkrg-kyextend-0.9.6-01.se.13.ky10.x86_64
  Verifying        : dkms-2.6.1-6.ky10.noarch
  Verifying        : lkrg-kyextend-0.9.6-01.se.13.ky10.x86_64

Installed:
  dkms-2.6.1-6.ky10.noarch                                  lkrg-kyextend-

Complete!
[root@localhost 桌面]#
```

```
# yum install libkydima
```

```
[root@localhost 桌面]# yum install libkydima
Last metadata expiration check: 0:06:43 ago on 2024年03月12日 星期二 13时49分54秒.
Dependencies resolved.
================================================================================
 Package          Architecture      Version                  Repository     Size
================================================================================
Installing:
 libkydima        x86_64            1.1.0-07.ky10            ks10-adv-os   322 k

Transaction Summary
================================================================================
Install  1 Package

Total download size: 322 k
Installed size: 2.0 M
Is this ok [y/N]: y
Downloading Packages:
libkydima-1.1.0-07.ky10.x86_64.rpm                  934 kB/s | 322 kB     00:00
--------------------------------------------------------------------------------
Total                                               931 kB/s | 322 kB     00:00
Running transaction check
Transaction check succeeded.
Running transaction test
Transaction test succeeded.
Running transaction
  Preparing        :                                                        1/1
  Installing       : libkydima-1.1.0-07.ky10.x86_64                         1/1
  Running scriptlet: libkydima-1.1.0-07.ky10.x86_64                         1/1
Created symlink /etc/systemd/system/multi-user.target.wants/kydima-init-policy.service → /usr/lib/systemd/system/kydima-init-policy.service.
Created symlink /etc/systemd/system/multi-user.target.wants/kydima-daemon.service → /usr/lib/systemd/system/kydima-daemon.service.

  Verifying        : libkydima-1.1.0-07.ky10.x86_64                         1/1

Installed:
  libkydima-1.1.0-07.ky10.x86_64

Complete!
[root@localhost 桌面]#
```

检测 auditd 服务是否正常启动：

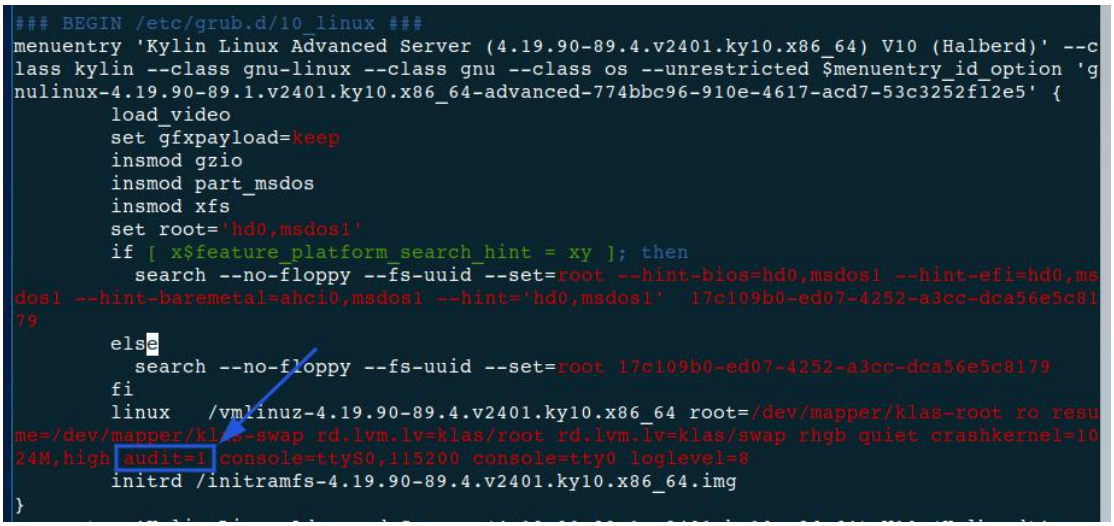
```
# systemctl status auditd
```



auditd 服务已启动，需要进一步检测内核 audit 审计是否开启，查看 grub 引导参数，是否配置启动了 audit 组件。

```
#cat /proc/cmdline
```

编辑 grub.cfg，将 audit 配置改成 audit=1，并重启系统。



也可以修改 /etc/default/grub 文件，启用内核 audit 审计。

```
[root@localhost etc]# cat default/grub
GRUB_TIMEOUT=5
GRUB_DISTRIBUTOR="$(sed 's, release .*$,,g' /etc/system-release)"
GRUB_DEFAULT=saved
GRUB_DISABLE_SUBMENU=true
GRUB_TERMINAL_OUTPUT="console"
GRUB_CMDLINE_LINUX="resume=/dev/mapper/klas-swap rd.lvm.lv=klas/root rd.lvm.lv=k
las/swap rhgb quiet crashkernel=1024M,high audit=0"
GRUB_DISABLE_RECOVERY="true"
[root@localhost etc]#
```

10.2.4.1.2. 功能开关配置

关闭：

```
# kydima_set switch -s off
```

开启：

```
# kydima_set switch -s on -c 5
```

10.2.4.1.3. 查看动态度量策略

```
[root@localhost ~]# kydima_get
用法: kydima_get
<switch|process|module|kernel|syscalls|idt|cycle|event> [-f] [-p] [-l]
功能: 进程度量与内核防护策略查询
参数:
    -h, --help 显示帮助信息
    switch:  开关策略
    cycle:   周期设置
    process: 进程度量
        -l, --list   查看列表
        -f, --file   文件路径
        -p, --pid    进程号
    module:  内核模块度量
```

```
        kernel:  内核度量
        syscalls: 系统调用表度量
        idt:     中断描述符表度量
        event:   事件触发开关
    pcr:  PCR 扩展策略
    举例:
        kydima_get process -f /usr/lib/systemd/systemd
        kydima_get process -p 1
        kydima_get module
        kydima_get kernel
        kydima_get syscalls
        kydima_get idt
        kydima_get switch
        kydima_get cycle
        kydima_get event
        kydima_get pcr
```

- 查看状态：

```
[root@localhost ~]# kydima_get switch
switch：on
```

- 查看周期度量时间：

```
[root@localhost ~]# kydima_get cycle
cycle：15 分钟
```

- 查看模块度量状态：

```
[root@localhost ~]# kydima_get module
module: on
```

- 查看进程度量信息：

```
[root@localhost ~]# kydima_get process /usr/lib/system/system
```

process: 进程路径：/usr/lib/systemd/systemd, 度量策略：仅审计, 度量结果：未篡改/已篡改

- 查看内核度量状态：

```
[root@localhost ~]# kydima_get kernel
kernel:  on
```

- 查看系统调用表度量状态：

```
[root@localhost ~]# kydima_get syscalls
syscalls:  on
```

- 查看中断描述符表度量状态：

```
[root@localhost ~]# kydima_get idt
idt:  on
```

- 查看事件触发开关状态：

```
[root@localhost ~]# kydima_get event
event：off
```

- 查看 PCR 扩展策略：

```
[root@localhost ~]# kydima_get pcr
pcr：off
pcr：register: 13
pcr：tpm :tpm2
```

- kydima_get 获取所有的进程策略被篡改的日志：

```
[root@localhost ~]# kydima_get process - list
```

10.2.4.1.4. 关于 pcr 配置注意事项

a）检测 tpm/tcm 设备节点是否存在

```
#ls /dev/t[pc]m*
```

tcm0 对应设备类型 tcm1

tcm2 对应设备类型 tcm2

tpm* 对应设备类型 tpm2

b）核实可信芯片是否为 pcie 外插 tcm 卡，该信息需要核实具体硬件无法从软件层面查询（当前系统驱动不支持，请勿使用）。

10.2.4.1.5. 配置动态度量策略

```
[root@localhost ~]# kydima_set
用法: kydima_set <switch|process|module|kernel|syscalls|idt|event> [-a]
[-d] [-f] [-p] [-s] [-c] <value>
功能：进程度量与内核防护策略设置
参数:
```

```
-h, --help 显示帮助信息
switch:    开关策略
    -s, --status   状态 on/off。其中，on 表示开启，off 表示关闭
                   当功能关闭时,无法配置其他参数
    -c, --cycle    周期设置(分钟)
process:   进程度量
    -a, --add   添加进程
    -d, --del   删除进程
    -v, --value   度量策略
        kill:   发现被篡改后杀死该进程
        audit:  发现被篡改后仅记录审计日志
    -f, --file        文件路径
    -p, --pid         进程号
module:    内核模块度量
    -s, --status   状态 on/off。其中，on 表示开启，off 表示关闭
kernel:    内核度量
    -s, --status   状态 on/off。其中，on 表示开启，off 表示关闭
syscalls: 系统调用表度量
    -s, --status   状态 on/off。其中，on 表示开启，off 表示关闭
idt:     中断描述符表度量
    -s, --status   状态 on/off。其中，on 表示开启，off 表示关闭
event:   事件触发开关
    -s, --status   状态 on/off。其中，on 表示开启，off 表示关闭
pcr:   PCR 扩展策略
      -s, --status   状态 on/off。其中，on 表示开启，off 表示关闭
        -r, --register 度量值扩展位置，默认使用 13，可取值 13~23
        -t, --tpm   指定硬件可信设备名称，如 tcm1、tcm2、tpm1、tpm2
```

举例:

```
kydima_set process -a -f /usr/lib/systemd/systemd -v audit
kydima_set process -d -f /usr/lib/systemd/systemd
kydima_set process -a -p 1 -v kill
```

```
kydima_set process -d -p 1
kydima_set module -s on
kydima_set kernel -s on
kydima_set syscalls -s on
kydima_set idt -s on
kydima_set switch -s on -c 5
kydima_set event -s on
kydima_set pcr -s on -r 13 -t tcm1
kydima_set pcr -s off
```

- 打开动态度量功能，并设置度量周期为 5 分钟

```
[root@localhost ~]# kydima_set switch -s on -c 5
```

- 添加一条策略，将 systemd 进程设置为发现被篡改后仅记录审计日志：

```
[root@localhost ~]# kydima_set process -a -f
/usr/lib/systemd/systemd -v audit
```

- 添加一条策略，将进程号为 1 的程序设置为发现被篡改后仅记录审计日志：

```
[root@localhost ~]# kydima_set process -a  -p  1 -v audit
```

- 打开对内核模块的度量

```
[root@localhost ~]# kydima_set module -s on
```

- 打开对内核的度量

```
[root@localhost ~]# kydima_set kernel -s on
```

- 打开对系统调用表的度量

```
[root@localhost ~]# kydima_set syscalls -s on
```

- 打开对中断描述符表的度量

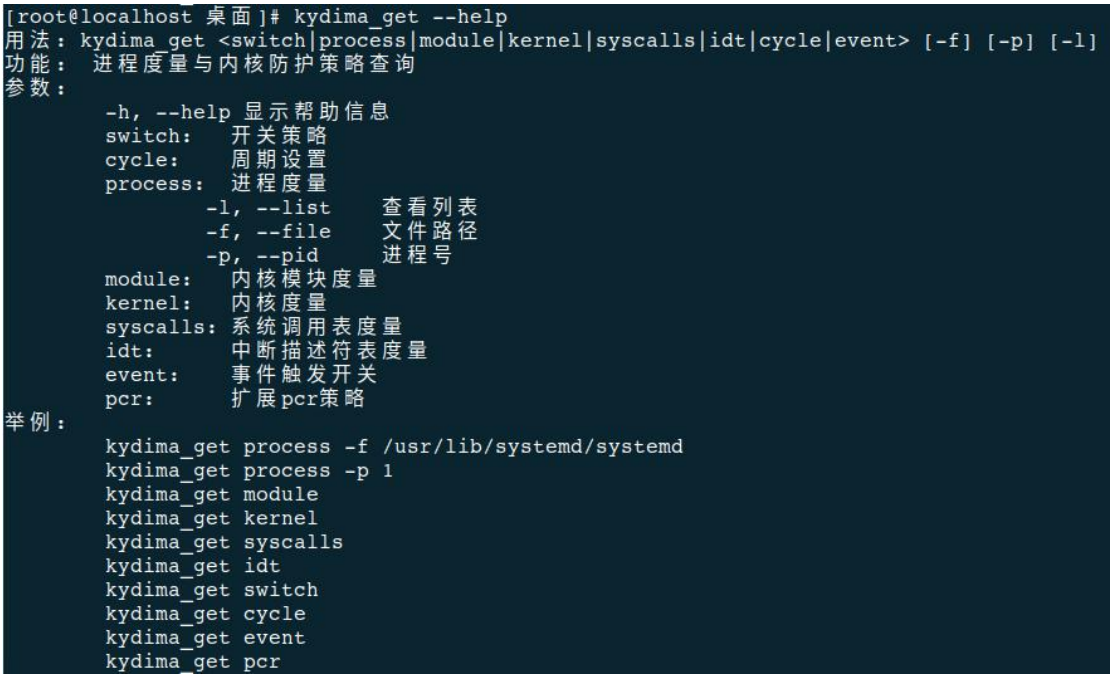
```
[root@localhost ~]# kydima_set idt  -s on
```

- 设置 PCR 扩展策略

```
[root@localhost ~]# kydima_set pcr -s on  # pcr 扩展功能开启
[root@localhost ~]# kydima_set pcr -s on -r 14 -t tcm1  # pcr 扩展开启，设置扩展到 14，采用 tcm1 可信设备
[root@localhost ~]# kydima_set pcr -s off  # pcr 扩展功能关闭
```

10.2.4.2. 示例

```
kydima_get --help
```




```
kydima_set --help
```

```
[root@localhost 桌面]# kydima_set --help
用法：kydima_set <switch|process|module|kernel|syscalls|idt|event> [-a] [-d] [-f] [-p] [-s] [-c] <value>
功能：  进程度量与内核防护策略设置
参数：
       -h, --help 显示帮助信息
       switch:    开关策略
              -s, --status   状态 on/off。其中，on表示开启，off表示关闭
                             当功能关闭时,无法配置其他参数
              -c, --cycle    周期设置(分钟)
       process:    进程度量
              -a, --add   添加进程
              -d, --del   删除进程
              -v, --value    度量策略
                     kill:    发现被篡改后杀死该进程
                     audit:   发现被篡改后仅记录审计日志
              -f, --file         文件路径
              -p, --pid          进程号
       module:    内核模块度量
              -s, --status   状态 on/off。其中，on表示开启，off表示关闭
       kernel:    内核度量
              -s, --status   状态 on/off。其中，on表示开启，off表示关闭
       syscalls: 系统调用表度量
              -s, --status   状态 on/off。其中，on表示开启，off表示关闭
       idt:      中断描述符表度量
              -s, --status   状态 on/off。其中，on表示开启，off表示关闭
       event:    事件触发开关
              -s, --status   状态 on/off。其中，on表示开启，off表示关闭
       pcr:    扩展pcr策略
              -s, --status   状态 on/off。其中，on表示开启，off表示关闭
              -r, --register   度量值扩展位置。默认使用13，可取值13~23
              -t, --tpm      指定硬件可信设备名称。如tcm1、tcm2、tpm2
举例：
       kydima_set process -a -f /usr/lib/systemd/systemd -v audit
       kydima_set process -d -f /usr/lib/systemd/systemd
       kydima_set process -a -p 1 -v kill
       kydima_set process -d -p 1
       kydima_set module -s on
       kydima_set kernel -s on
       kydima_set syscalls -s on
       kydima_set idt -s on
       kydima_set switch -s on -c 5
       kydima_set event -s on
       kydima_set pcr -s on
       kydima_set pcr -s on -r 14 -t tpm2
       kydima_set pcr -s off
```

kydima 基本操作：

```
[root@localhost 桌面]# kydima_set switch -s off
[root@localhost 桌面]# kydima_get switch
kyrg模块未加载！！！
[root@localhost 桌面]# kydima_set switch -s on -c 5
[root@localhost 桌面]# kydima_get switch
switch:  on
[root@localhost 桌面]# kydima_set module -s on
[root@localhost 桌面]# kydima_get switch
switch:  on
[root@localhost 桌面]# kydima_get module
module:  on
[root@localhost 桌面]# kydima_set kernel -s on
[root@localhost 桌面]# kydima_get kernel
kernel:  on
[root@localhost 桌面]# kydima_set syscalls -s on
[root@localhost 桌面]# kydima_get syscalls
syscalls:  on
[root@localhost 桌面]# kydima_set idt  -s on
[root@localhost 桌面]# kydima_get idt
idt:  on
[root@localhost 桌面]#
```

```
[root@localhost 桌面]# kydima_set kernel -s off
[root@localhost 桌面]# kydima_get kernel
kernel:  off
[root@localhost 桌面]# kydima_set module -s off
[root@localhost 桌面]# kydima_get module
module:  off
[root@localhost 桌面]# kydima_set syscalls -s off
[root@localhost 桌面]# kydima_get syscalls
syscalls:  off
[root@localhost 桌面]# kydima_set idt -s off
[root@localhost 桌面]# kydima_get idt
idt:  off
```

配置进程监控策略配置：

```
# kydima_set process -a -f /usr/lib/systemd/systemd -v audit
# kydima_set process -a  -p  1 -v audit
```

```
[root@localhost ~]#  kydima_set process -a -f /usr/lib/systemd/systemd -v audit
[root@localhost ~]#  kydima_set process -a  -p  1 -v audit
[root@localhost ~]# kydima_get process  -l
       id |进程路径                                                pid     度量结果        度量策略

       1 |/usr/lib/systemd/systemd                                                       仅审计(audit)
         /usr/lib/systemd/systemd                                 2348    未篡改          仅审计(audit)
         /usr/lib/systemd/systemd                                 2346    未篡改          仅审计(audit)
         /usr/lib/systemd/systemd                                 1       未篡改          仅审计(audit)
```

10.2.5. 核外安全功能及配置

10.2.5.1. 安全切换工具

security-switch 是系统下提供的一个用于安全配置的工具，通过该工具可进行如下 3 种安全模式的切换：

- default 模式：启用系统 kysec 安全机制；
- strict 模式：启用 kysec、selinux、三权分立，且 selinux 为 ukmcs 策略；
- none 模式: 关闭 kysec、selinux、三权分立。

10.2.5.1.1. default 模式

设置 default 模式

1.root 登录系统，执行：

$security-switch --set default

2.重启系统

模式状态：

```
------------------------------------------------------------------
 安全模块    |    当前状态    | 默认状态
------------------------------------------------------------------
Kysec       启用(Enforcing)      启用(Enforcing)
```

10.2.5.1.2. strict 模式

设置 strict 模式

1.root 登录系统，执行：

security-switch --set strict

（然后分别设置 root,secadm,auditadm 三个管理员用户的密码）

2.重启系统

模式状态

```
------------------------------------------------------------------
 安全模块    |    当前状态    | 默认状态
------------------------------------------------------------------
SELinux          启用(Enforcing)    启用(Enforcing)
三权分立         启用               启用
执行控制         启用(Enforcing)      启用(Enforcing)
```

10.2.5.1.3. 关闭模式

设置关闭模式

1.root 登录系统，设置关闭模式：

#security-switch --set none

2.重启系统；

模式状态

```
当前安全级别：关闭(none)
--------------------------------------------------
```

```
 安全模块     |     当前状态     |     默认状态
-----------------------------------------------------
```

10.2.5.2. 审计系统权限管理

基于基础安全审计系统功能，麒麟操作系统根据三权分立要求，实现管理员分权机制，修订只允许审计管理员具有审计服务管理权限和审计规则修改权限。根据其系统安全机制要求，增加审计服务容错机制，当审计服务出错时，审计服务将自动重启，确保审计服务正常运行；移除 SU 到审计管理员的权限，限制系统管理员通过 SU 命令执行 auditctl 等审计管理工具。同时，添加了不同等级的审计规则、修复在启用或不启用三权分立时对用户的判断等功能。

审计系统权限需要在安全状态为 strict 模式时相应功能才会开启，另外要确保系统的审计服务处于开启状态（systemctl statsu auditd）。

10.2.5.2.1. 登录审计

审计用户登录时，无论成功与否，均会在审计日志中有信息输出。审计日志包括事件类型、时间、用户、事件成功与否、身份鉴别请求的源等，如：

```
Authentication Report
============================================
#date time acct host term exe success event
============================================
2023 年 09 月 07 日 13:21:21 test localhost localdomain /dev/tty1
/usr/libexec/gdm-session-worker yes 309
```

10.2.5.2.2. 行为审计

添加用户审计规则后，用户的行为会被系统审计。

示例（如对 uid 为 600 的用户添加行为审计规则）：

```
1.审计管理员添加审计规则，对 uid=600 的用户行为进行审计：
#auditctl -a exit,always -S all -F uid=600
2.安全管理员执行操作：
#cat /etc/uid_list
3.审计管理员以 uid 进行审计信息查看
```

```
#ausearch -ui 600 | grep uid_list
```

此时可在在审计日志中查看到针对步骤 2 的审计信息。

10.2.5.2.3. 账号管理审计

所有用户账号的增加、删除（useradd、passwd、userdel）等行为都会被系统审计。

示例如下，在审计用户下通过$aureport -m 可以查看到用户账户操作的审计日志。

```
1.root 创建一个用户：
#useradd testuser
2.root 修改用户密码：
#passwd testuser
3.root 删除用户
#userdel testuser
```

10.2.5.2.4. 文件、目录监视审计

添加文件、目录监视规则后，任何对文件、目录的操作都会被系统审计。

示例如下：

```
存在 kylin 用户创建的/tmp/file1 文件及/tmp/test.dir 目录：
1.审计管理员登录系统，添加审计规则：
$auditctl -w /tmp/file1 -k file
$auditctl –w /tmp/test.dir –k test_dir
2.kylin 用户 cat 查看/tmp/file1 文件
3.kylin 用户使用 rm 删除/tmp/file1 文件
4.kylin 用户对/tmp/test.dir 进行读操作
5.kylin 用户对/tmp/test.dir 进行写操作
6.审计管理员登录系统，删除步骤 1 中审计规则：
$auditctl -W /tmp/file1 -k file
$auditctl –W /tmp/test.dir –k test_dir
```

步骤 1 添加规则后，步骤 2、3、4、5 等对于文件及目录的操作均会在审计日志中体现；步骤 6 删除规则后不再记录相关操作审计日志。

10.2.5.2.5. 规则配置

添加排除规则后可以排除指定的审计信息。

10.2.5.2.6. 审计日志保护

只有审计管理员才可以查看审计日志。

10.2.5.2.7. 审计日志转储

当审计日志达到设置的最大值时，会覆盖所存储的最早的审计记录。该部分需要审计管理员对配置文件/etc/audit/auditd.conf 进行手动编辑。

示例：

```
1.审计管理员编辑/etc/audit/auditd.conf 文件，使 max_log_file=1，
num_logs = 3 即日志文件最大为 1MB，备份文件数为 3
2.审计管理员重启服务
#systemctl restart auditd
3.审计管理员添加审计规则，
$auditctl -a exit,always -S all -Fsys
4.安全管理员随便执行任何操作，以便产生大量日志
$find /
5.审计管理员查看审计日志文件大小和数目
$ls -lh /var/log/audit
```

经步骤 5 查看，系统审计日志有：audit.log、audit.log.1、audit.log.2，其中 audit.log.1、audit.log.2 文件大小都为 1M 左右。

10.2.5.2.8. 审计日志超阈值警报

当审计跟踪的磁盘空间已到达极限时，有审计报警日志提醒。对于磁盘空间极限值的设定，需要审计用户手动调整/etc/audit/auditd.conf。

示例，假设系统日志分区总大小 50G，将 30G 设置为剩余空间阈值，即当用系统日志占用 20G 以上时会有审计日志产生：

```
1.审计管理员编辑/etc/audit/auditd.conf 文件，使 space_left=30000，即
日志磁盘空间阈值为 30G；space_left_action=SYSLOG，即向日志中写入一
条报警日志；（具体磁盘空间阀值根据测试系统实际日志磁盘空间确定）
2.审计管理员重启服务
#systemctl restart auditd
```

步骤 2 后，如果日志空间剩余容量小于 30G，则审计日志中将产生审计日志信息。

10.2.5.2.9. 支持安全机制审计日志类别

目前审计服务会根据系统中不同的安全机制功能写入不同类型的审计日志类型。

如 kysec 类的审计日志类型为：

```
type=KYSEC_STATUS msg=audit(1692350159.191:22): status=4
old_status=4  loginuid=4294967295 uid=0 gid=0 session=4294967295
res=warn]UID="root" GID="root"
type=KYSEC_AVC msg=audit(1692350631.711:248): denied
{ execute } for pid=5595 comm='bash' name='/root/hello.sh'
ssid=0x0b osid=0x00 loginuid=0 session=2 kysec_status=2 res=warn
```

11. 机密计算

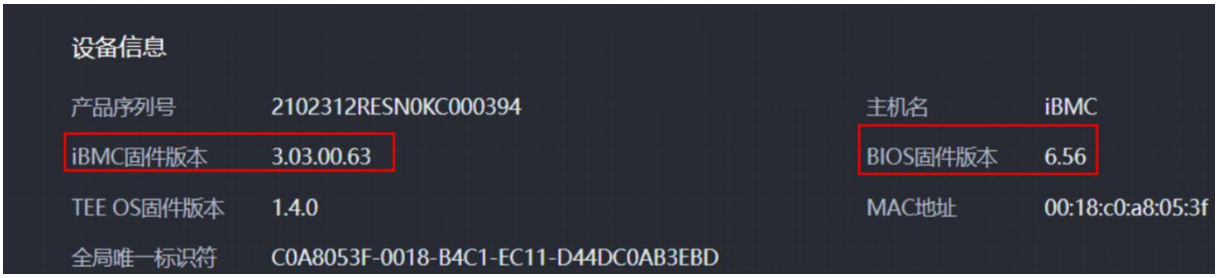
11.1. 鲲鹏

11.1.1. 软硬件环境确认

在着手软硬件环境的确认流程之前，至关重要的一环是对鲲鹏服务器进行预先配置的核查，以确保 TrustZone 安全套件已被正确预置。为此，应仔细参照附录 1 中详述的检查指南，以验证服务器是否已具备这一关键的安全特性。

11.1.1.1. 检查 iBMC、BIOS 版本

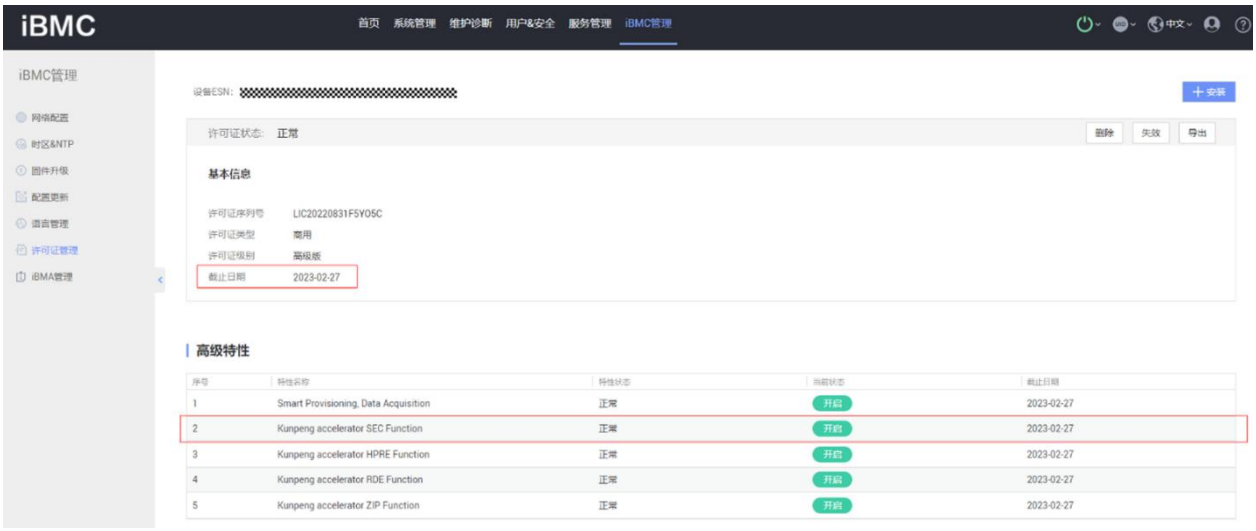
登录 iBMC，在首页可查看 iBMC、BIOS 固件版本号。



要求 iBMC 固件版本不低于 3.01.12.49，BIOS 固件版本不低于 1.83。如果有任意一项不满足则视为服务器不支持鲲鹏 TrustZone 套件。

11.1.1.2. 检查 TrustZone License

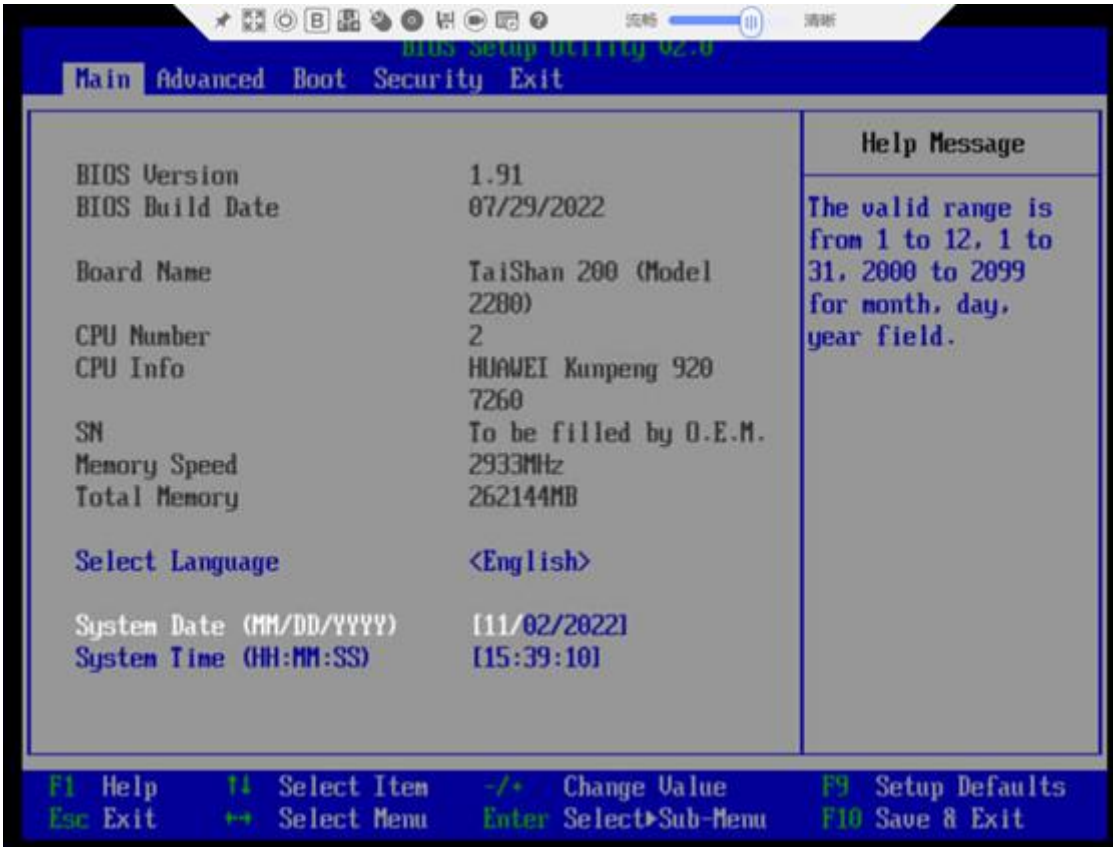
登录 iBMC，在首页依次单击“iBMC 管理->许可证管理”，查看 License 加载情况。



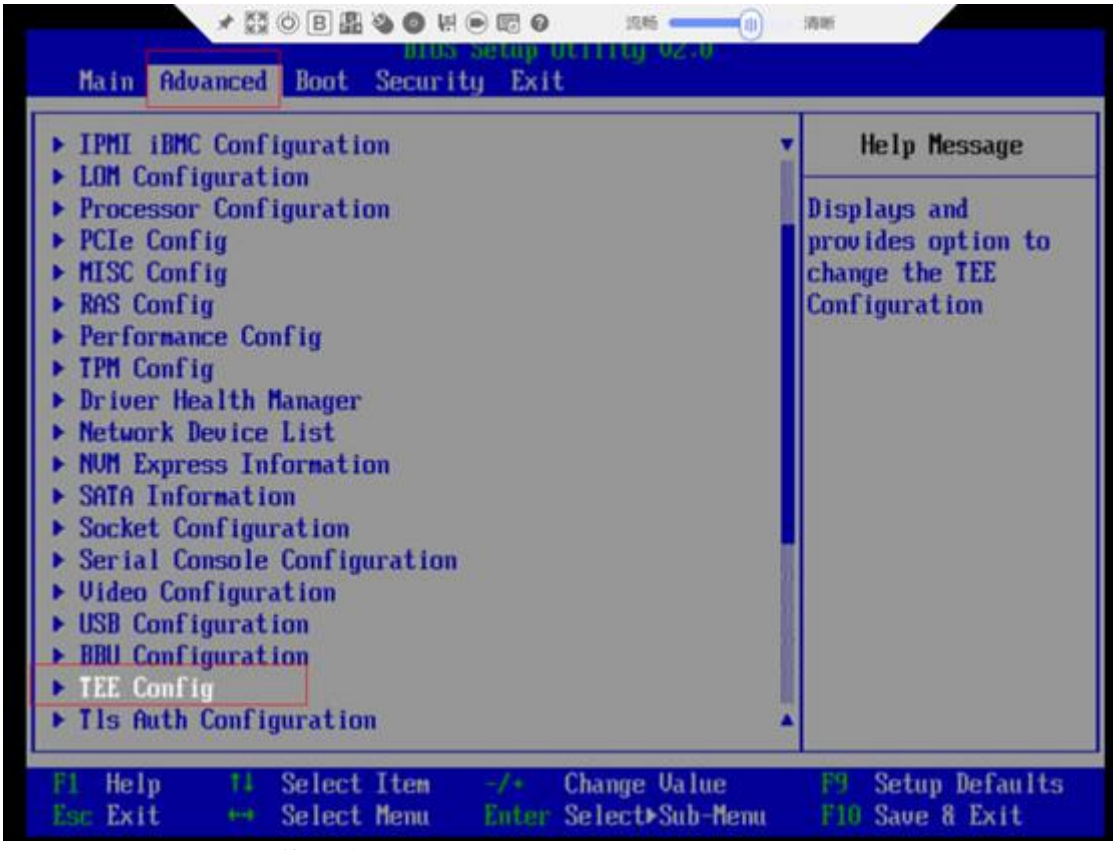
要求许可证已导入，许可证仍在有效期内且“Kunpeng accelerator SEC Function”高级特性处于“开启”状态。如果不满足，即使服务器已经烧写 TrustZone 套件相关固件，也无法使能鲲鹏 TrustZone 功能。

11.1.1.3. 检查安全 OS 启动密钥

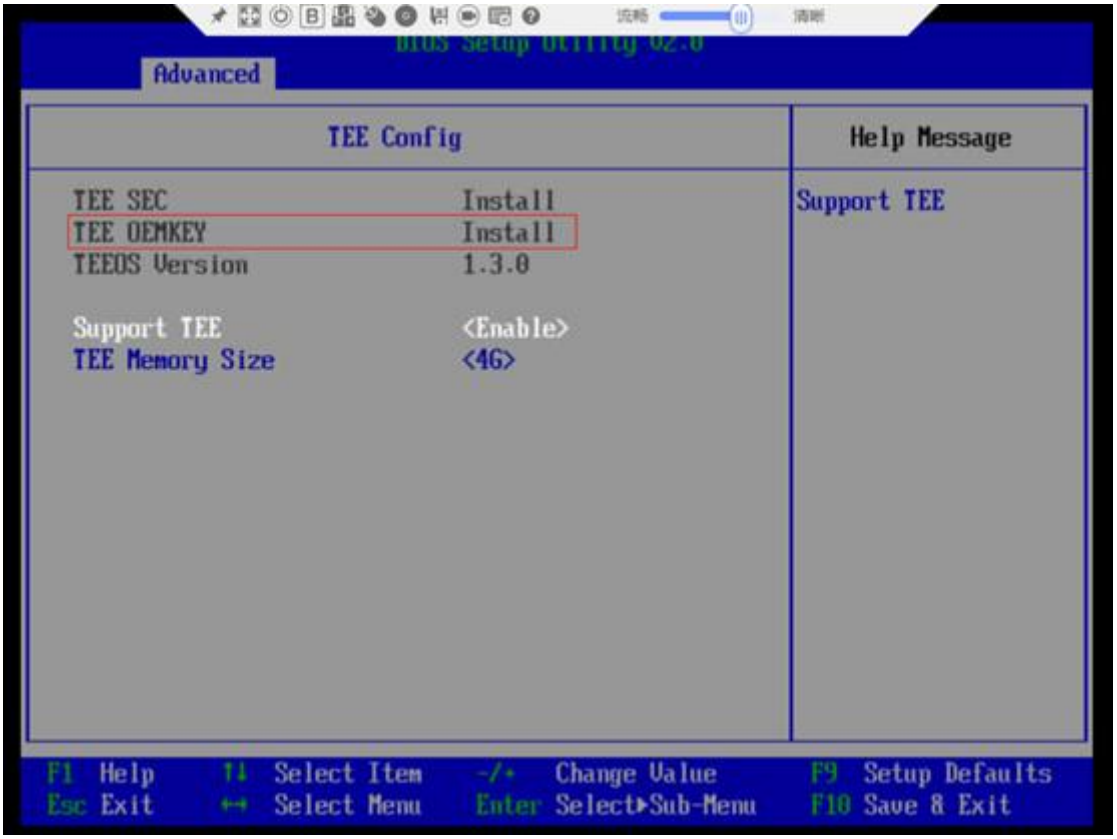
登录服务器 BIOS。



依次选择“Advanced->TEE Config”查看 TEE 配置选项。



查看 OEMKEY 安装状态。



如果显示“TEE OEMKEY”处于“Install”状态，至此该服务器已预置鲲鹏 TrustZone 套件，可通过配置“Support TEE”使能鲲鹏服务器 TrustZone 功能。

11.1.2. 软件包安装

CA 应用需要 REE 侧 patch 才能实现与 TEE 侧的 TA 应用通信，REE 侧 patch 与 TEE OS 固件包存在版本配套关系，请参考下表（来自：https://www.hikunpeng.com/document/detail/zh/kunpengcctrustzone/fg-tz/kunpengtrustzone_20_0019.html）。

TEE OS hpm固件版本	itrustee_tzdriver配套代码tag	itrustee_client配套代码tag
1.1.0 <= version < 1.2.0	v1.0.0	v1.0.0
1.2.0	v1.2.0	v1.2.0
1.3.0 <= version < 1.4.0	v1.3.0	v1.3.0
1.4.0	master	master

需根据 BMC 中显示的 TEE OS 固件版本，选择安装对应的 itrsutee_tzdriver、itrustee_client 软件包，比如下面以 1.2.0 为例：


```
[root@DC5-09-004 ~]# rpm -qa | grep itrustee
itrustee_tzdriver_v1.2.0-1.0.0-01.aarch64
itrustee_client_v1.2.0-1.0.0-01.aarch64
[root@DC5-09-004 ~]#
```

11.1.3. 部署及运行

11.1.3.1. 加载 REE 侧内核模块

进入“/opt/itrustee_tzdriver_版本号”目录，执行：

```
#./deploy.sh
#lsmod | grep tzdriver
```

```
[root@DC5-09-004 ~]# cd /opt/itrustee_tzdriver_v1.2.0/
[root@DC5-09-004 itrustee_tzdriver_v1.2.0]# ./deploy.sh
------------------------------------------------------------
准备开始部署iTrustee tzdriver
请确认TEE OS固件版本为1.2.0

是否开始部署？（Y/y:是，N/n:退出）> y

部署结束
------------------------------------------------------------
[root@DC5-09-004 itrustee_tzdriver_v1.2.0]# lsmod | grep tzdriver
tzdriver              208896  0
[root@DC5-09-004 itrustee_tzdriver_v1.2.0]#
```

（说明：tzdriver.ko 加载后不支持卸载，仅支持服务器下电再启动的方式将其恢复为未加载状态）

11.1.3.2. 启动 REE 侧 teecd 守护进程

进入“/opt/itrustee_client_版本号”目录，执行：

```
#./deploy.sh
#nohup /usr/bin/teecd &
#ps aux | grep teecd
```

```
[root@DC5-09-004 ~]# cd /opt/itrustee_client_v1.2.0/
[root@DC5-09-004 itrustee_client_v1.2.0]# ./deploy.sh
------------------------------------------------------------
准备开始部署iTrustee client
请确认TEE OS固件版本为1.2.0

是否开始部署？（Y/y:是，N/n:退出）> y

部署结束
------------------------------------------------------------
[root@DC5-09-004 itrustee_client_v1.2.0]# nohup /usr/bin/teecd &
[1] 3158
[root@DC5-09-004 itrustee_client_v1.2.0]# nohup: ignoring input and appending output to 'nohup.out'

[root@DC5-09-004 itrustee_client_v1.2.0]# ps aux | grep teecd
root        3158  0.6  0.0 103884  3684 pts/0    Sl   11:59   0:00 /usr/bin/teecd
root        3164  0.0  0.0   5960  1768 pts/0    S+   11:59   0:00 grep --color=auto teecd
[root@DC5-09-004 itrustee_client_v1.2.0]#
```

（说明：teecd 必须以绝对路径运行，即“/usr/bin/teecd”，“&”符号表示后台执行）

12. 安全启动

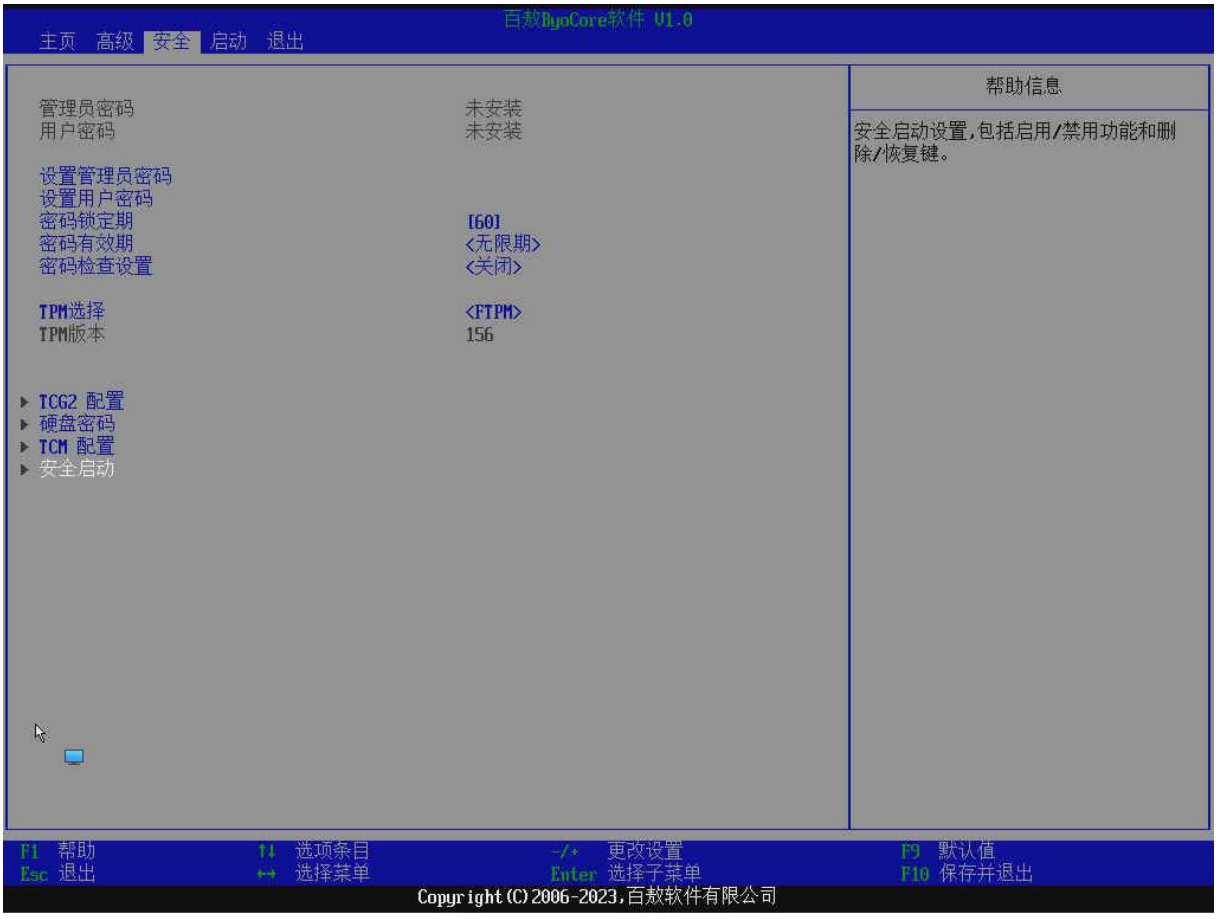
12.1. 概述

安全启动是 PC 行业成员开发的一种安全标准，用于帮助确保设备仅使用受原始设备制造商 (OEM) 信任的软件进行启动。 当 PC 启动时，固件会检查每个启动软件片段的签名，如果签名有效，则电脑将会启动，而固件会将控制权转递给操作系统。

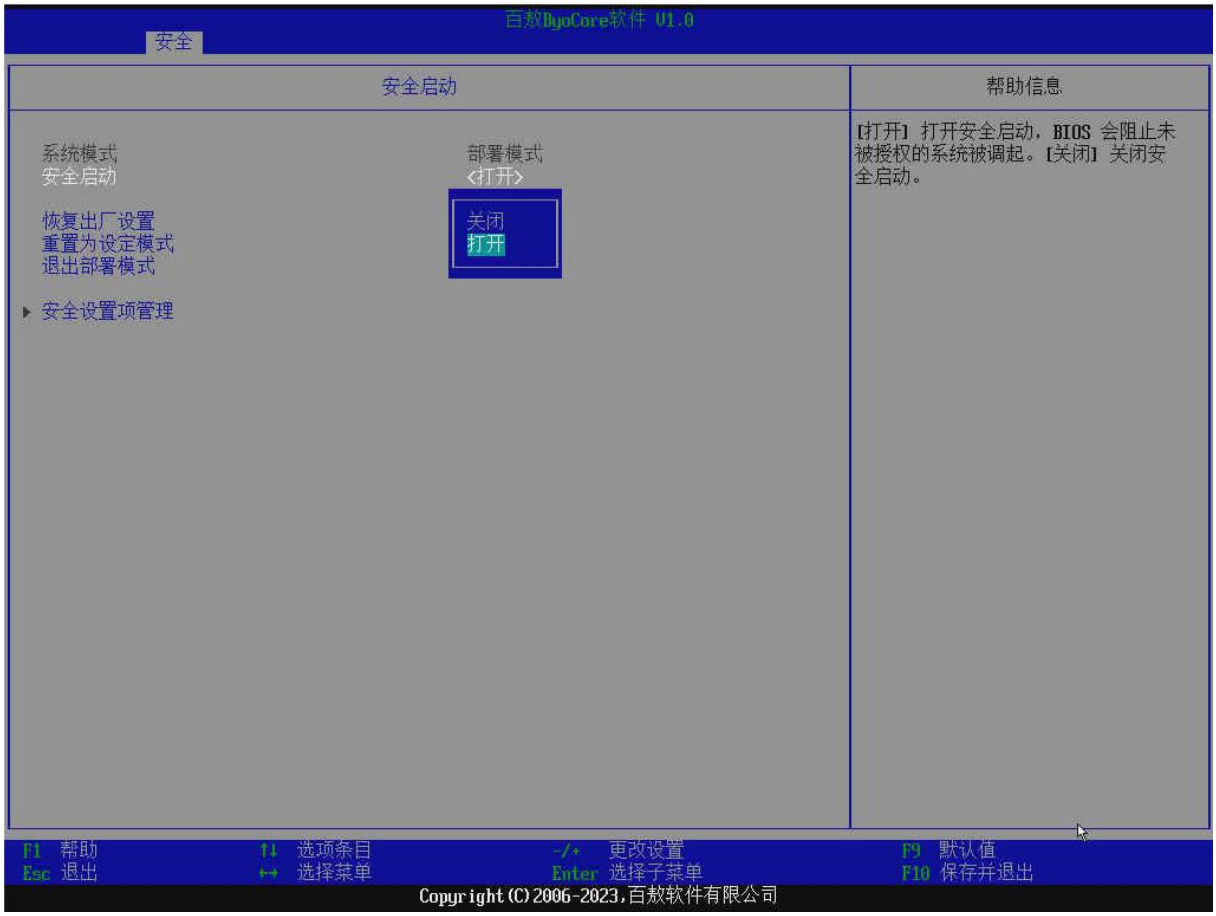
12.2. 物理机安全启动

物理机安全启动需要固件的支持，可以进入固件查看是否支持安全启动功能，并且开启安全启动后，固件中需要导入麒麟安全启动签名证书才能进入系统，否则将启动失败。

进入固件界面，选安全，有安全启动选项，如下图：



选择安全启动，将可以选择开启或者关闭安全启动，如下图：



选择下面安全设置管理，将进入固件证书数据库，如下图：

选择 DB 选项，选择进入，将进入证书管理界面，如下图：

选择登记证书将进入证书查看和导入界面，如下图：

开启安全启动，导入麒麟证书后，重启，可以正常进入系统。

12.3. 虚拟机安全启动

虚拟机开启安全启动需要下载 edk2-ovmf 软件包，并对虚拟机进行配置。使用 virsh edit 编辑虚拟机配置文件，在<os>标签中添加以下内容。

x86 虚拟机：

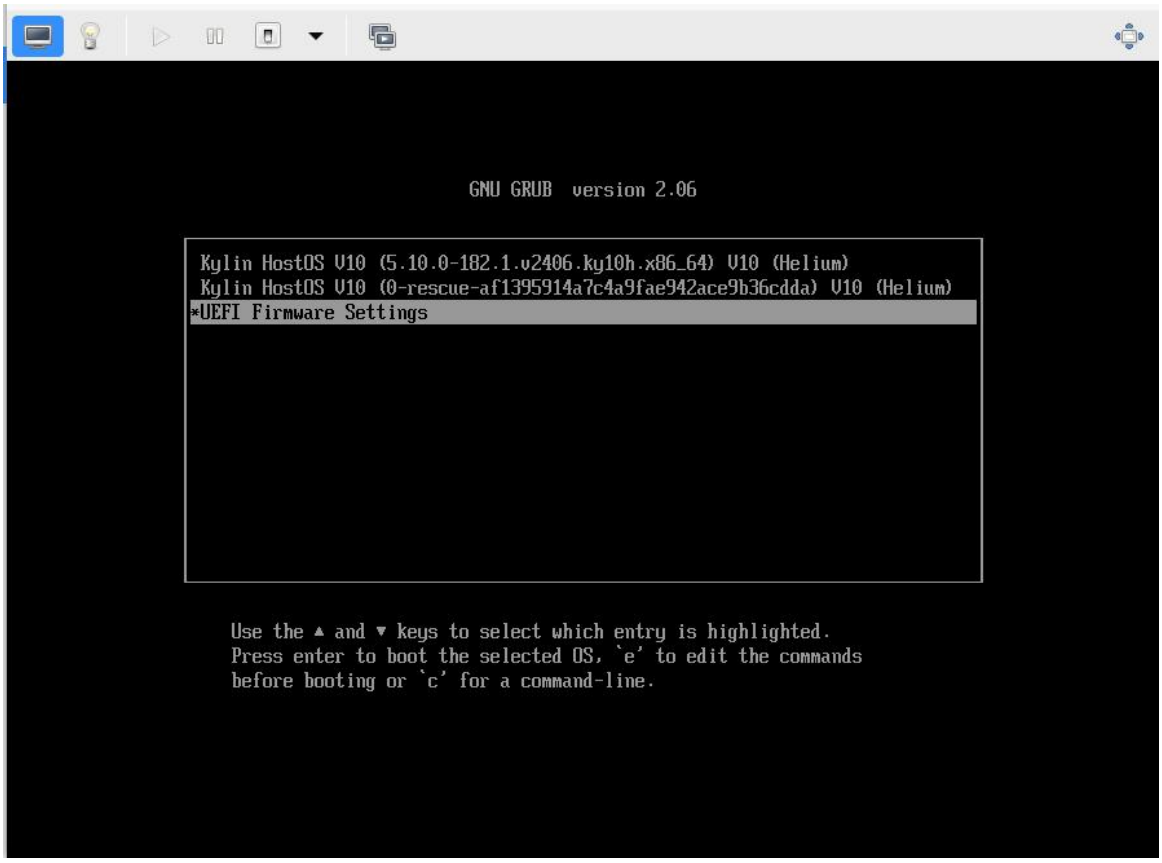
```
#virsh edit –domain <虚拟机名称>
…
<os>

…

<loader readonly='yes'
type='pflash'>/usr/share/edk2/ovmf/OVMF_CODE.fd</loader>

<nvram
template='/usr/share/edk2/ovmf/OVMF_VARS_SECUREBOOT.fd'>/path/to/虚拟
```

```
机名称_VARS.fd</nvram>
    </os>
    …
```

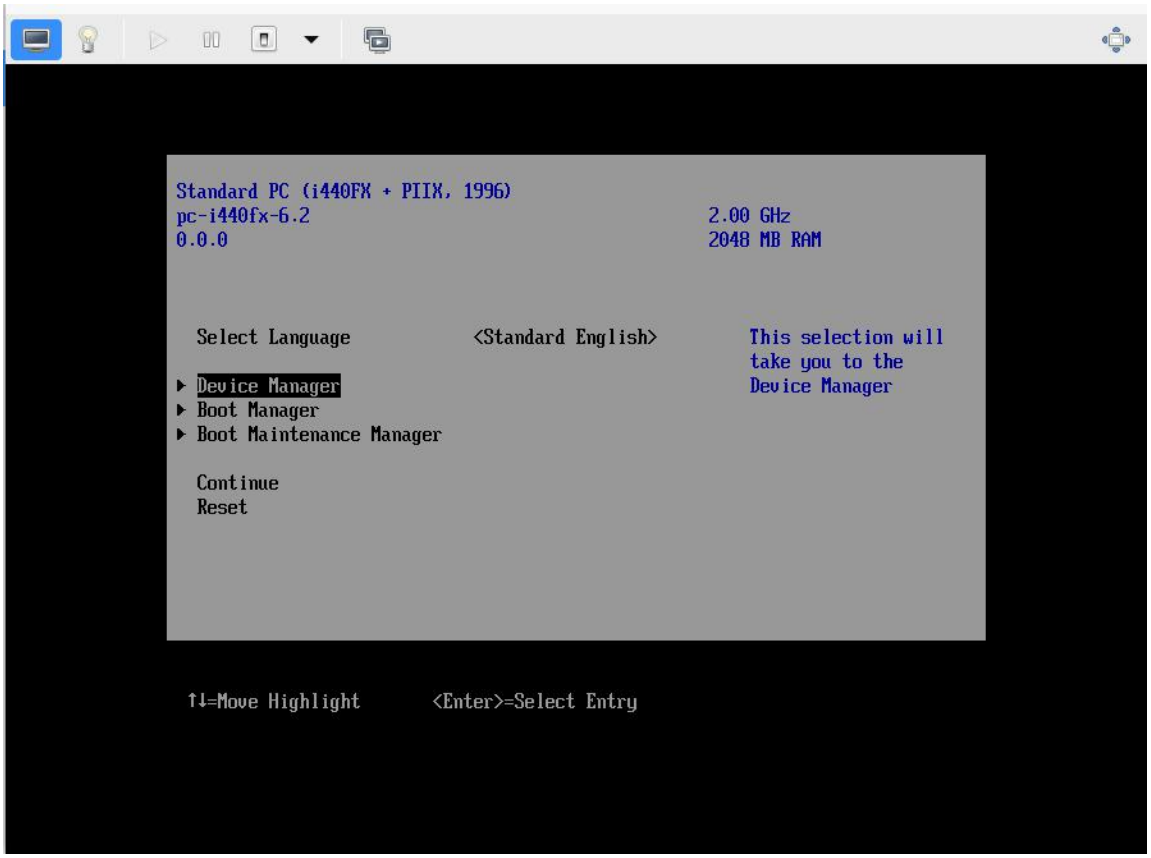
ARM 虚拟机：

```
    #virsh edit –domain <虚拟机名称>
    …
    <os>
    …
    <loader readonly='yes'
type='pflash'>/usr/share/edk2/aarch64/QEMU_EFI-pflash,raw</loader>
    <nvram template='
/usr/share/edk2/aarch64/vars-template-secureboot-pflash.fd'>/path/to/虚拟机名称
_VARS.fd</nvram>
    </os>
    …
```

配置完成后便可以安装虚拟机系统。系统安装完成后，进入 UEFI 设置界面，确认安全启动已开启，在 grub 界面选择 UEFI Firmware Settings 进入设置界面。

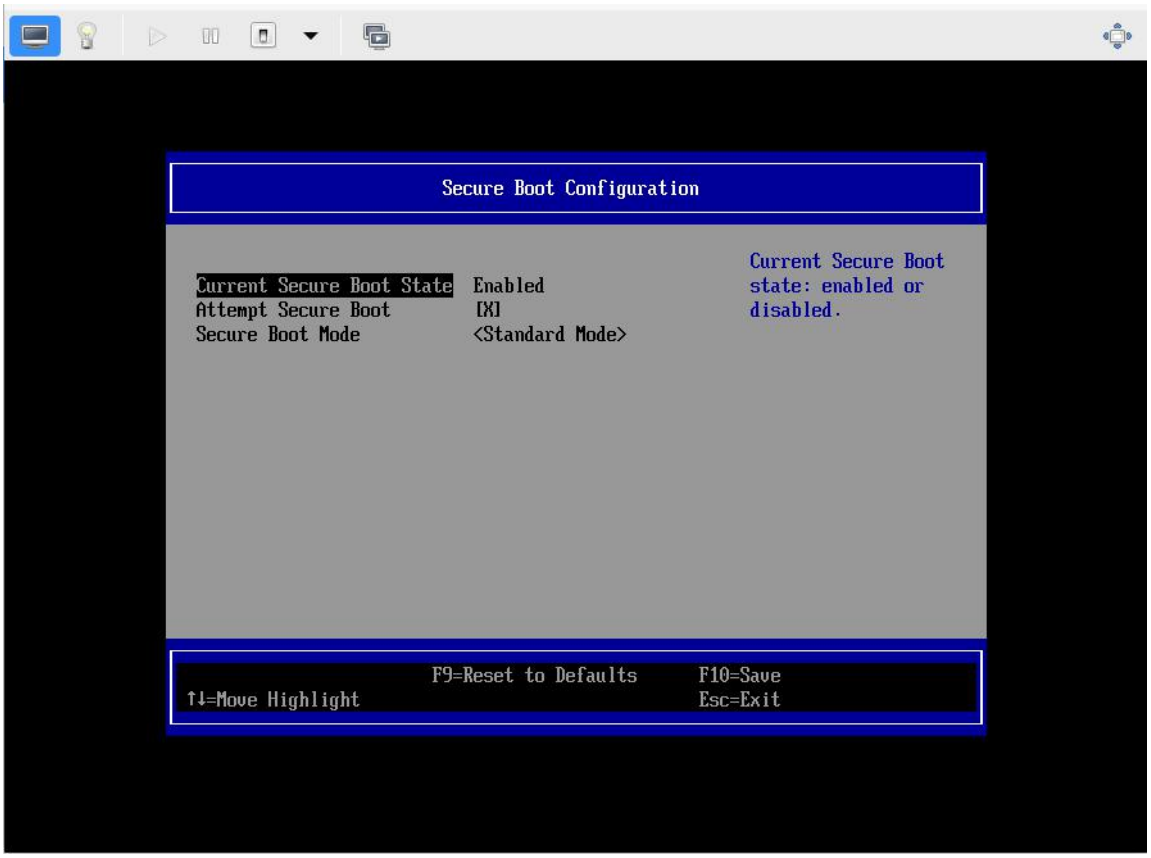


选择 Device Manager 进入设备管理界面。

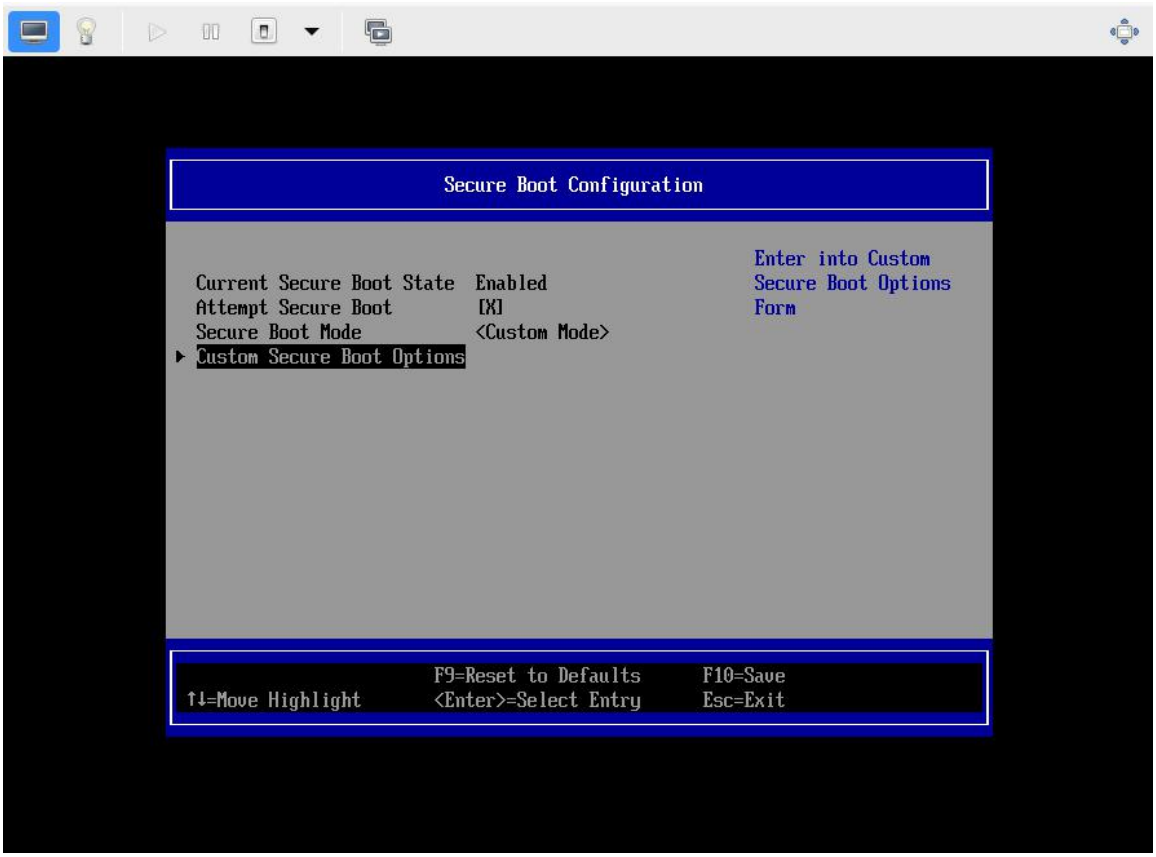


之后，选择 Secure Boot Configuration 进入安全启动管理界面。

如果 Attempt Secure Boot 已选中，则说明已经开启安全启动。如果想要关闭安全启动，取消选中该选项即可。



如果需要更改或添加 PK、KEK 等证书，将 Secure Boot Mode 切换为 Costom Mode，然后选择 Custom Secure Boot Option，进入自定义安全启动选项界面。



13. DPU 算力卸载

13.1. 概述

在数据中心及云场景下，随着摩尔定律失效，通用处理单元 CPU 算力增长速率放缓，而同时网络 IO 类速率及性能不断攀升，二者增长速率差异形成的剪刀差，即当前通用处理器的处理能力无法跟上网络、磁盘等 IO 处理的需求。传统数据中心下越来越多的通用 CPU 算力被 IO 及管理面等占用，这部分资源损耗称之为数据中心税（Data-center Tax）。据 AWS 统计，数据中心税可能占据数据中心算力的 30%以上，部分场景下甚至可能更多。

DPU 的出现就是为了将这部分算力资源从主机 CPU 上解放出来，通过将管理面、网络、存储、安全等能力卸载到专有的处理器芯片（DPU）上进行处理加速，达成降本增效的结果。目前主流云厂商如 AWS、阿里云、华为云都通过自研芯片完成管理面及相关数据面的卸载，达成数据中心计算资源 100%售卖给客户。

管理面进程卸载到 DPU 可以通过对组件源码进行拆分达成，将源码根据功能逻辑拆分成独立运行的两部分，分别运行在主机和 DPU，达成组件卸载的目的。但是这种做法有以下问题：一是影响组件的软件兼容性，组件后续版本升级和维护需要自己维护相关 patch，带来一定的维护工作量；二是卸载工作无法被其他组件继承，后续组件卸载后仍需要进行代码逻辑分析和拆分等工作。为解决上述问题，本方案提出 DPU 的无感卸载，通过 OS 提供的抽象层，屏蔽应用在主机和 DPU 间跨主机访问的差异，让业务进程近似 0 改动达成卸载到 DPU 运行的目标，且这部分工作属于操作系统通用层，与上层业务无关，其他业务进行 DPU 卸载时也可以继承。

13.2. qtfs 共享文件系统

13.2.1. 基础介绍

qtfs 是一个共享文件系统项目，可部署在 host-dpu 的硬件架构上，也可以部署在

host-vm 上，通过 vsock 建立安全通信通道。以客户端服务器的模式工作，使客户端能通过 qtfs 访问服务端的指定文件系统，就像访问本地文件系统一样。qtfs 的特性：

- 支持挂载点传播；
- 支持 proc、sys、cgroup 等特殊文件系统的共享；
- 客户端对 qtfs 目录下文件的操作都被转移到服务端，文件读写可共享；
- 支持在客户端对服务端的文件系统进行远程挂载；
- 可以定制化处理特殊文件；
- 支持远端 fifo、unix-socket 等，并且支持 epoll，使客户端和服务端像本地通信一样使用这些文件；
- 基于 host-dpu 架构时，底层通信方式可以支持 PCIe，性能大大优于网络；
- 内核模块形式开发，无需对内核进行侵入式修改。

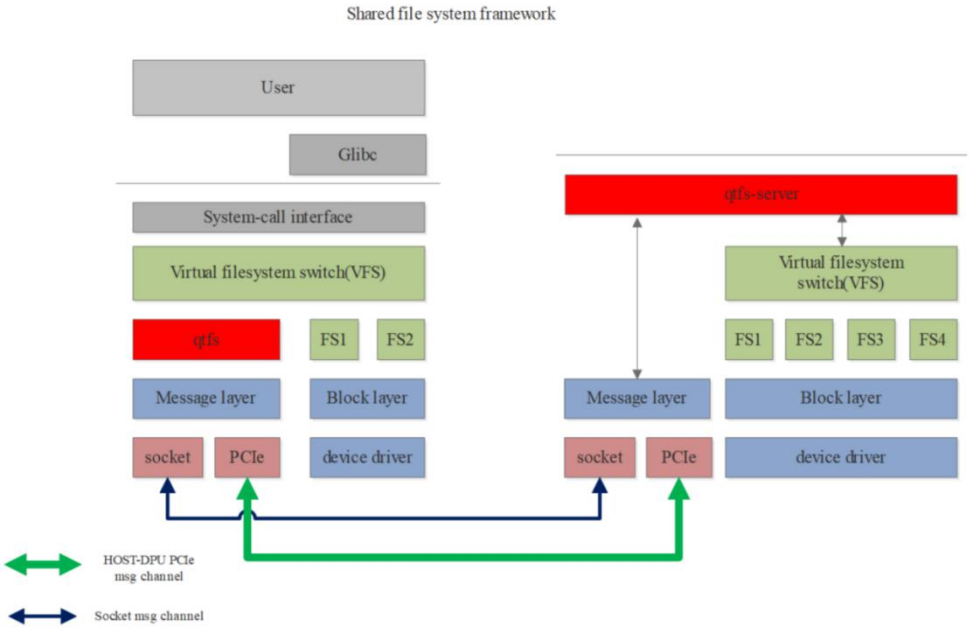


图 12-1 qtfs 软件架构

13.2.2. vsock 模式 qtfs 部署

如有 DPU 硬件支持通 vsock 与 host 通信，可选择此方法。 如果没有硬件，也可以选择 host-vm 作为 qtfs 的 client 与 server 进行模拟测试，通信通道为 vsock。以下为 host-vm 的配置方式。

1. 环境配置

服务端为物理机 host，客户端为 host 上的虚拟机，且虚拟机要与物理机网络互通，以下的虚拟机为 NAT 网络。选择 host(172.30.241.233)作为 qtfs 的 server 端，vm(192.168.122.53)为 qtfs 的 client 端，通信模式为 vhost-vsock。

2. 配置软件源

```
#  cat /etc/yum.repos.d/kylin_x86_64.repo
...
[v10-hostos-2309-epkl]
name = v10-hostos-2309-epkl
baseurl =
http://xxxx/kojifiles/repos/v10-hostos-2309-epkl-build/latest/x86_64/
gpgcheck = 0
enable =1
...
```

3. 关闭防火墙和 selinux，host 和 vm 均需操作

```
# systemctl stop firewalld
# systemctl disable firewalld
# setenforce 0
# sed -i 's#SELINUX=enforcing#SELINUX=disabled#g' /etc/selinux/config
```

13.2.2.1. host 部署 qtfs-server

首先部署 qtfs-server 端，等待 qtfs-client 端连接。

1. 安装软件包 qtfs-server

```
# yum install qtfs-server
```

2. 加载 vhost_vsock 模块

vhost_vsock 必须在 vm 启动前插入，否则 vm 无法正常使用 vsock。

```
# modprobe vhost_vsock
```

```
[root@localhost ~]# lsmod | grep vsock
vhost_vsock            24576  0
vhost                  61440  1 vhost_vsock
vsock_loopback         16384  0
vmw_vsock_virtio_transport_common    49152  2 vhost_vsock,vsock_loop
vmw_vsock_vmci_transport    40960  0
vsock                  53248  8 vmw_vsock_virtio_transport_common,vh
vmw_vmci               94208  1 vmw_vsock_vmci_transport
```

3. 加载 qtfs_server.ko 模块

由于内核版本一直在升级，软件仓库中的 qtfs_server.ko 的编译时内核和安装时内核可能不一致，因此做了内核 ko 模块与内核版本解耦的修改，qtfs_server.ko 存在于以下两个路径中的一个。

- 编译内核和安装内核一致，则进入此路径

```
# cd /lib/modules/$(uname -r)/extra
```

- 编译内核和安装内核不一致，则进入此路径

```
# cd /lib/modules/$(uname -r)/weak-updates
```

注：qtfs_server.ko 被软链接到此目录。

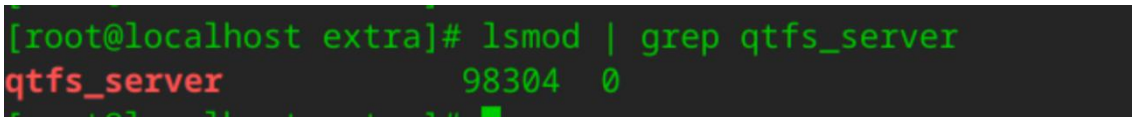
- 加载 qtfs_server.ko

```
# insmod qtfs_server.ko qtfs_server_vsock_cid=2
qtfs_server_vsock_port=12345 qtfs_log_level=WARN
```

注：host 的 cid 值配置为 2，所以 qtfs_server_vsock_cid=2，代码中定义 qtfs_server_vsock_port=12345

- 查看加载是否成功

```
# lsmod | grep qtfs_server
```



4. 开放 whitelist，共享 qtfs-server 端文件系统

/etc/qtfs/whitelist 是只读权限，所以需要先修改该文件权限，用户按需添加开放的文件目录权限，这里以/home/VMs 为例。

```
# mkdir /home/VMs
# chmod 600 /etc/qtfs/whitelist
# cat /etc/qtfs/whitelist
[Open]
Path=/home/VMs;/var/lib/libvirt/qemu
[Write]
Path=/home/VMs;/var/lib/libvirt/qemu
[Read]
Path=/home/VMs;/var/lib/libvirt/qemu
[Readdir]
Path=/home/VMs;/var/lib/libvirt/qemu
```

```
[Mkdir]
Path=/home/VMs;/var/lib/libvirt/qemu
[Rmdir]
Path=/home/VMs;/var/lib/libvirt/qemu
[Create]
Path=/home/VMs;/var/lib/libvirt/qemu
[Unlink]
Path=/home/VMs;/var/lib/libvirt/qemu
[Rename]
Path=/home/VMs;/var/lib/libvirt/qemu
[Setattr]
Path=/home/VMs;/var/lib/libvirt/qemu
[Setxattr]
Path=/home/VMs;/var/lib/libvirt/qemu
[Mount]
Path=/home/VMs;/var/lib/libvirt
```

5. 创建 qtfs 运行时目录

```
# mkdir /var/run/qtfs
```

6. 启动 engine

```
# nohup engine 16 1 172.30.241.233 12121 192.168.122.53 12121 2>&1
&
```

注：16 表示 engine 的线程数；1 表示 udsproxy 的线程数

- 查看是否正确启动 engine

```
[root@localhost zmx]# ps -ef | grep engine
root      161046  137900 99 10:47 pts/2    00:05:03 engine 16 1 172.30.241.233 12121 192.168.122.53 12121
[root@localhost zmx]# ll /var/run/qtfs/
总用量 0
-rw-------. 1 root root 0  8月  7 10:47 engine.lock
srwxrwxrwx. 1 root root 0  8月  7 10:47 remote_uds.sock
-rw-------. 1 root root 0  8月  7 10:47 uds.lock
```

- qtfs_server 模块此时已经被使用

```
[root@localhost zmx]# lsmod | grep qtfs_server
qtfs_server            98304  1
```

13.2.2.2. vm 部署 qtfs-client

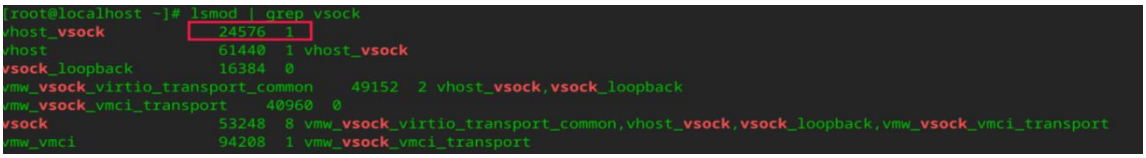
1. 创建 vm

启动 vm 时为 vm 配置为 vsock 通道，参考配置如下，将 vsock 段加在 devices 配置内：

```
# virsh edit vm
  <devices>
......
    <vsock model='virtio'>
      <cid auto='no' address='10'/>
</vsock>
......
  </devices>
```

注：这里的 cid 值需要根据配置决定，因为 host 作为 server 端，host 的 cid 值配置为 2，vm 的 cid 需要配置成其他的值，这里 cid=10。

2. vm 启动后查看 host 上 vhost_sock 是否被使用



注：如果红框中”1”的位置为”0”，则需要 virsh destroy vm，重启 vm。

3. 安装软件包 qtfs-client

```
#  yum install qtfs-client
```

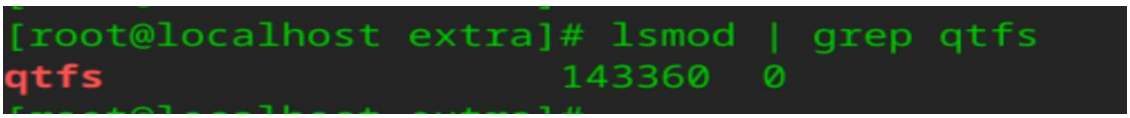
4. 加载 qtfs.ko 模块

同前节 qtfs_server.ko，请进入到对应的目录下。

- 加载 qtfs.ko

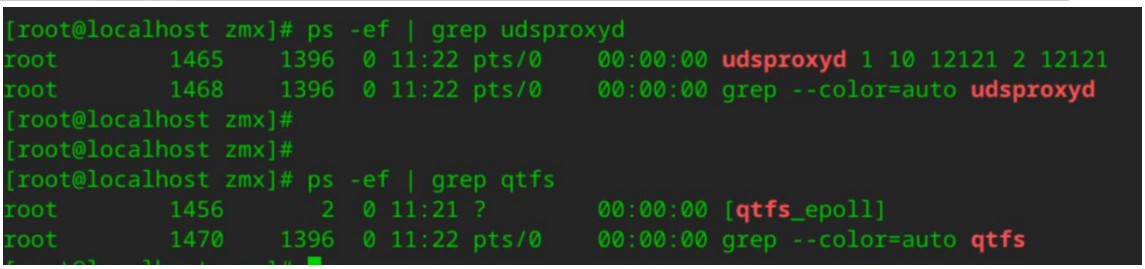
```
#  insmod qtfs.ko qtfs_server_vsock_cid=2 qtfs_server_vsock_port=12345 qtfs_log_level=WARN
```

- 查看加载是否成功



5. 启动 udsproxyd

```
#  nohup udsproxyd 1 10 12121 2 12121 2>&1 &
```



注：1 代表 udsproxyd 的进程数，10 代表 vm 的 cid，2 代表 host 的 cid。

13.2.3. vsock 模式 qtfs 功能验证

1. 挂载验证

- 在 vm qtfs-client 中创建目录（whitelist 中开放的权限）

```
#  mkdir -p /root/mnt/home/VMs
```

- 客户端通过挂载把服务端的文件系统让客户可见

```
# mount -t qtfs /home/VMs /root/mnt/home/VMs
```

- server 端创建文件 1111，在 client 端可见文件 1111(反之亦然)

server 端

```
[root@localhost VMs]# ll
总用量 4
-rw-r--r--. 1 root root 5  8月   7 11:30 1111
```

client 端

```
[root@localhost VMs]# ll
总用量 4
-rw-r--r--. 1 root root 5  8月   7 11:30 1111
[root@localhost VMs]# pwd
/root/mnt/home/VMs
[root@localhost VMs]#
```

2. 开源文件系统测试集 pjdfstest

pjdfstest 是一个 POSIX 系统接口的测试套，用于进行文件系统接口兼容性测试。

- vm 中安装 pjdfstest

```
# git clone https://gitee.com/mengxuanzhang/pjdfstest.git
# cd pjdfstest
# autoreconf -ifs
# ./configure
# make pjdfstest
```

- 执行测试

```
# cd /root/mnt/home/VMs
# prove -rv ${pjdfstest 文件所在目录}
```

```
Test Summary Report
-------------------
/root/pjdfstest/tests/mkdir/06.t          (Wstat: 0 Tests: 8 Failed: 0)
  Parse errors: Bad plan.  You planned 12 tests but ran 8.
Files=198, Tests=3049, 145 wallclock secs ( 1.06 usr  0.26 sys + 12.53 cusr 14.62 csys = 28.47 CPU)
Result: FAIL
```

13.3. 虚拟化管理面无感卸载

13.3.1. 基础介绍

虚拟化管理面，即 libvirtd，而虚拟化管理面卸载，即是将 libvirtd 卸载到虚拟机所在机器（以下称为 host）之外的另一台机器（以下称为 DPU）上运行。

我们使用了 qtfs 将 host 的一些与虚拟机运行相关的目录挂载到 DPU 上，使得虚拟化管理面工具可以访问到这些目录，为 kvm 虚拟机准备运行所需要的环境，此处，因为需要挂载远端的 proc 和 sys，所以，我们创建了一个专门的 rootfs 以作为 libvirtd 的运行环境（以下称为/another_rootfs）。

并且通过 rexec 执行虚拟机的拉起、删除等操作，使得可以将虚拟化管理面和虚拟机分离在不同的两台机器上，远程对虚拟机进行管理。

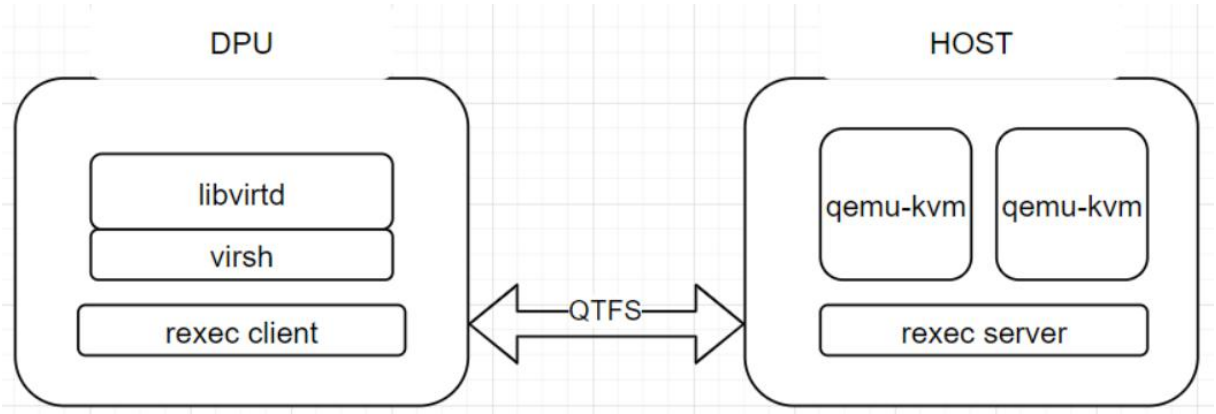


图 12-2 libvirt 卸载架构图

- rexec 介绍

rexec 是一个用 c 语言开发的远程执行组件，分为 rexec client 和 rexec server。server 端为一个常驻服务进程，client 端为一个二进制文件，client 端被执行后会基于 udsproxyd 服务与 server 端建立 uds 连接，并由 server 常驻进程在 server 端拉起指定程序。在 libvirt 虚拟化卸载中，libvirtd 卸载到 DPU 上，当它需要在 HOST 拉起虚拟机 qemu 进程时调起 rexec client 进行远程拉起。

13.4. 容器管理面无感卸载

13.4.1. 基础介绍

容器管理面，即 kubernetes、dockerd、containerd、isulad 等容器的管理工具，而容器管理面卸载，即是将容器管理面卸载到与容器所在机器（以下称为 HOST）之外的另一台机器（当前场景下是指 DPU，一个具备独立运行环境的硬件集合）上运行。

我们使用共享文件系统 qtfs 将 HOST 上与容器运行相关的目录挂载到 DPU 上，使得容器管理面工具（运行在 DPU）可以访问到这些目录，并为容器（运行在 HOST）准备运行所需要的环境，此处，因为需要挂载远端的 proc 和 sys 等特殊文件系统，所以，我们创建了一个专门的 rootfs 以作为 kubernetes、dockerd 的运行环境（以下称为/another_rootfs）。

并且通过 rexec 执行容器的拉起、删除等操作，使得可以将容器管理面和容器分离在不同的两台机器上，远程对容器进行管理。

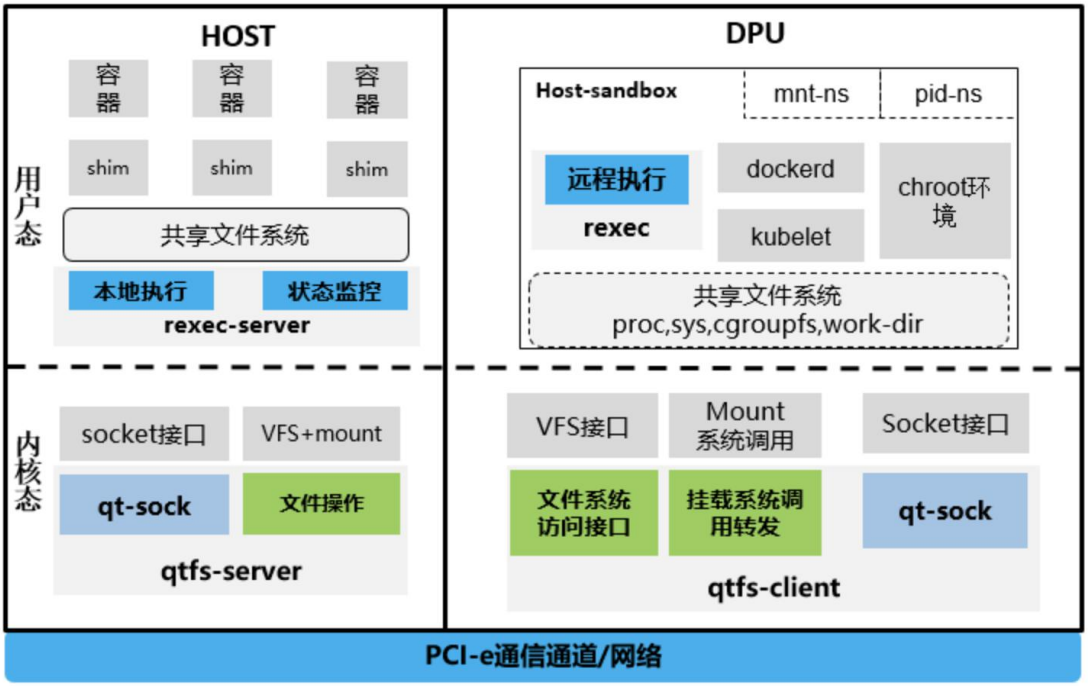


图 12-3 容器管理面 DPU 无感卸载架构

- dockerd 相关改动介绍

对 dockerd 的改动基于 18.09 版本。

在 containerd 中，暂时注释掉了通过 hook 调用 libnetwork-setkey 的部分，此处

不影响容器的拉起。并且，为了 docker load 的正常使用，注释掉了在 mounter_linux.go 中 mount 函数中一处错误的返回。

最后，因为在容器管理面的运行环境中，将/proc 挂在了服务端的 proc 文件系统，而本地的 proc 文件系统则挂载在了/local_proc，所以，dockerd 以及 containerd 中的对/proc/self/xxx 或者/proc/getpid()/xxx 或者相关的文件系统访问的部分，我们统统将/proc 改为了/local_proc。

- containerd 相关改动介绍

对于 containerd 的改动基于 containerd-1.2-rc.1 版本。

在获取 mountinfo 时，因为/proc/self/mountinfo 只能获取到 dockerd 本身在本地的 mountinfo，而无法获取到服务端的 mountinfo，所以，将其改为了/proc/1/mountinfo，使其通过获取服务端 1 号进程 mountinfo 的方式得到服务端的 mountinfo。

在 contaienrd-shim 中，将与 containerd 通信的 unix socket 改为了用 tcp 通信，containerd 通过 SHIM_HOST 环境变量获取 containerd-shim 所运行环境的 ip，即服务端 ip。用 shim 的哈希值计算出一个端口号，并以此作为通信的端口，来拉起 containerd-shim.

并且，将原来的通过系统调用给 contaienr-shim 发信号的方式，改为了通过远程调用 kill 指令的方式向 shim 发信号，确保了 docker 杀死容器的行为可以正确的执行。

- kubernetes 相关改动介绍

kubelet 暂不需要功能性改动，可能会遇到容器 QoS 管理器首次设置失败的错误，该错误不影响后续 Pods 拉起流程，暂时忽略该报错。

14. UKUI 安装指南

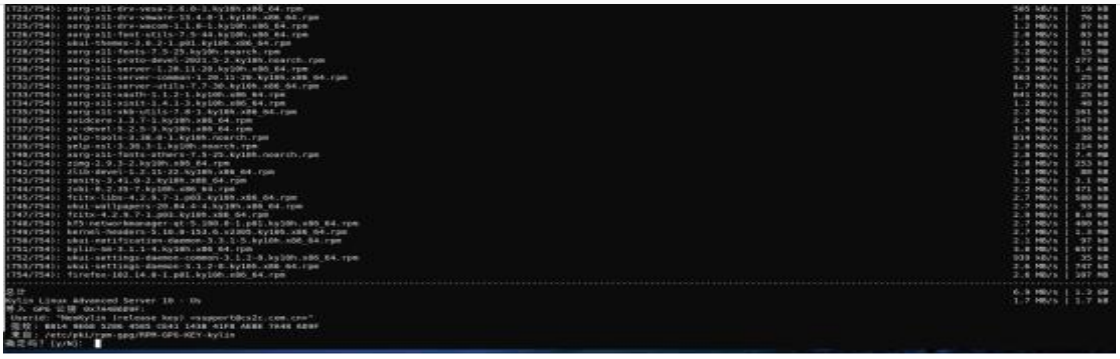
UKUI 是一款轻量级的桌面环境，Hostos 默认情况下是没有提供相关功能的，如果您想在服务器中使用 UKUI GUI，可以按照以下小节内容进行配置。

执行 yum 配置（镜像默认源）

```
root@localhost ~]# yum clean all
10 文件已删除
[root@localhost ~]# yum makecache
Kylin Linux Advanced Server 10 - Os
3.6 MB/s |  16 MB     00:04
Kylin Linux Advanced Server 10 - Updates
473 kB/s | 308 kB     00:00
上次元数据过期检查：0:00:01 前，执行于 2023 年 09 月 12 日 星期二 16 时 55 分 58 秒。
元数据缓存已建立。
```

安装带 GUI 的软件包，由于 HostOS，未对 UKUI 核心组件包做包分组管理，因此需要逐一安装。请按照如下命令执行。

```
# yum install cups ukui-enable-manual-login ukui-server fcitx fcitx-qt5
fcitx-configtool kylin-screenshot kylin-display-switch kylin-calculator  firefox
peony-share peony-device-rename pluma time-shutdown
xorg-x11-fonts-others ukui-notification-daemon mate-terminal
mesa-dri-drivers
```



安装完成后您可以使用以下命令来将默认运行级别更改为图形化用户界面：

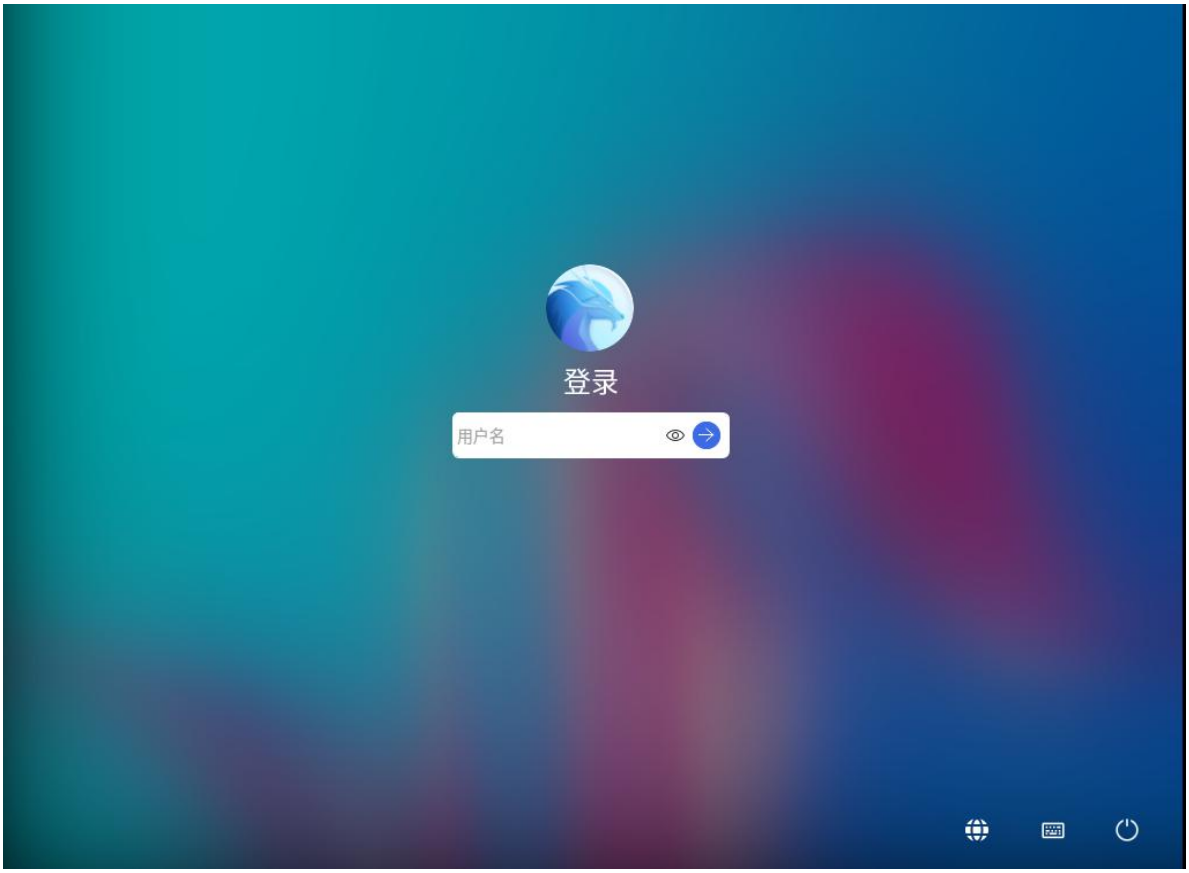
```
# systemctl set-default graphical.target
```

```
[root@localhost ~]# systemctl set-default graphical.target
Removed /etc/systemd/system/default.target.
Created symlink /etc/systemd/system/default.target → /usr/lib/systemd/system/graphical.target.
[root@localhost ~]#
```

重启系统:

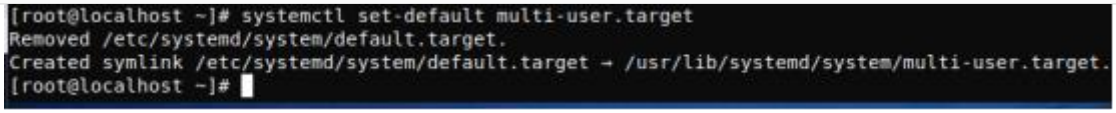
```
# reboot
```

此时您的系统已切换为图形化用户界面，如下图:



如果您想将图形化界面重新切换为命令行界面可通过以下命令:

```
# systemctl set-default multi-user.target
```



重启系统后即可切换成功:

```
# reboot
```

15. 二次集成工具

15.1. 镜像制作工具 oemaker

15.1.1. oemaker 工具介绍

oemaker 是用于制作 DVD 光盘映像的构建工具。通过该工具,用户可以基于 kylin 仓库构建 DVD 镜像。

15.1.2. 软硬件要求

使用构建工具 oemaker 制作 DVD 光盘所使用的机器需要满足如下软硬件要求：

- CPU 架构为 aarch64 或者 x86_64
- 操作系统为 银河麒麟云底座操作系统 V10
- 建议预留 30 GB 以上的磁盘空间（用于运行构建工具和存放 ISO 镜像）

15.1.3. 安装工具

1、已安装银河麒麟云底座操作系统 V10

2、使用 root 权限，安装镜像制作工具

```
# sudo yum install -y oemaker
```

3、使用 root 权限，确认工具是否安装成功

```
# sudo oemaker -h
```

15.1.4. 制作 DVD 镜像

15.1.4.1. oemaker 命令介绍

DVD 光盘映像构建工具通过 oemaker 命令执行功能。命令的使用格式为：

oemaker [-h] [-t Type] [-p Product] [-v Version] [-r RELEASE] [-s REPOSITORY]

oemaker 参数说明表如下表。

参数	说明
-t	镜像类型，可选参数为 standard debug source everything everything_debug everything_src 和 netinst（目前仅支持 standard）
-p	产品名，例如，Kylin
-v	版本号
-r	发行信息
-s	dnf 源地址
-h	帮助信息

15.1.4.2. 软件包来源

DVD 构建工具 -s 参数指定的 dnf 源地址为软件包仓库，例如：https://update.cs2c.com.cn/NS/V10/HPC/os/adv/lic/base/x86_64/。

15.1.4.3. 操作指导

说明：

- 如果对系统安装时分组有定制需求，请编辑修改 /opt/oemaker/config/`uname -m`/normal.xml。
- oemaker 的所有操作需要使用 root 权限。
- 目前仅支持 standard 镜像制作。

1、获取麒麟官方镜像源

2、确定系统的磁盘空间大于 30 GB 。

3、执行 DVD 光盘映像构建工具

场景：制作 DVD 光盘映像

```
    # sudo oemaker -p "Kylin" -r 'hostos' -v 'V10' -t standard -s
https://update.cs2c.com.cn/NS/V10/HPC/os/adv/lic/base/x86_64/
```

15.2. 镜像定制裁剪工具 isocut

15.2.1. isocut 工具简介

在某些场景下，用户不需要安装镜像提供的全量软件包，或者需要一些额外的软件包。因此，kylin 提供了镜像裁剪定制工具。通过该工具，用户可以基于 kylin 光盘镜像裁剪定制仅包含所需 RPM 软件包的 ISO 镜像。这些软件包可以来自原有 ISO 镜像，也可以额外指定，从而满足用户定制需求。

本章节介绍镜像裁剪定制工具 isocut 的安装和使用方法，以指导用户更好的完成镜像裁剪定制。

15.2.2. 软硬件要求

使用 裁剪定制工具 isocut 制作 ISO 所使用的机器需要满足如下软硬件要求：

- CPU 架构为 aarch64 或者 x86_64
- 操作系统为 银河麒麟云底座操作系统 V10
- 建议预留 30 GB 以上的磁盘空间（用于运行裁剪定制工具和存放 ISO 镜像）

15.2.3. 安装工具

1、已安装银河麒麟云底座操作系统 V10

2、使用 root 权限，安装镜像定制裁剪工具

```
    # sudo yum install -y isocut
```

3、使用 root 权限，确认工具是否安装成功

```
    # sudo isocut -h
```

15.2.4. 裁剪定制镜像

15.2.4.1. isocut 命令介绍

镜像裁剪定制工具通过 isocut 命令执行功能。命令的使用格式为：

isocut [--help | -h] [-t <temp_path>] [-r <rpm_path>] [-k <file_path>] < source_iso > < dest_iso >

isocut 参数说明表如下表。

参数	是否必选	参数含义
--help | -h	否	查询命令的帮助信息。
-t <temp_path>	否	指定工具运行的临时目录 temp_path，其中 temp_path 为绝对路径。默认为 /tmp 。
-r <rpm_path>	否	用户需要额外添加到 ISO 镜像中的 RPM 包路径。
-k <file_path>	否	用户需要使用 kickstart 自动安装，指定 kickstart 模板路径。
source_iso	是	用于裁剪的 ISO 源镜像所在路径和名称。不指定路径时，默认当前路径。
dest_iso	是	裁剪定制生成的 ISO 新镜像存放路径和名称。不指定路径时，默认当前路径。

表 142

15.2.4.2. 软件包来源

新镜像的 RPM 包来源有：

➢ 原有 ISO 镜像。该情况通过配置文件 /etc/isocut/rpmlist 指定需要安装的 RPM 软件包，配置格式为 "软件包名.对应架构"，例如：kernel.aarch64 。

➢ 额外指定。执行 isocut 时使用 -r 参数指定软件包所在路径，并将添加的 RPM 包按上述格式添加到配置文件 /etc/isocut/rpmlist 中。

➢ 说明：

➢ 裁剪定制镜像时，若无法找到配置文件中指定的 RPM 包，则镜像中不会添加该 RPM 包。

➢ 若 RPM 包的依赖有问题，则裁剪定制镜像时可能会报错。

15.2.4.3. kickstart 功能介绍

用户需要实现镜像自动化安装，可以通过 kickstart 的方式。在执行 isocut 时使用 -k 参数指定 kickstart 文件。

isocut 为用户提供了 kickstart 模板，路径是 /etc/isocut/anaconda-ks.cfg，用户可以基于该模板修改。

15.2.4.4. 操作指导

说明：

- 请不要修改或删除 /etc/isocut/rpmlist 文件中的默认配置项。
- isocut 的所有操作需要使用 root 权限。

1、修改配置文件 /etc/isocut/rpmlist，指定用户需要安装的 RPM 软件包

2、确定运行镜像裁剪定制工具的临时目录空间大于 8 GB 。

3、执行裁剪定制

1）场景一：新镜像的所有 RPM 包来自原有 ISO 镜像

```
# sudo isocut -t /home/temp
/home/isocut_iso/Kylin-Host-V10-General-Release-x86_64-Build05-20230906.iso
/home/result/new.iso
```

2）场景二：新镜像的 RPM 包除来自原有 ISO 镜像，还包含来自 /home/rpms 的额外软件包

```
# sudo isocut -t /home/temp -r /home/rpms
/home/isocut_iso/Kylin-Host-V10-General-Release-x86_64-Build05-20230906.iso
/home/result/new.iso
```

3）场景三：使用 kickstart 文件实现自动化安装，需要修改 /etc/isocut/anaconda-ks.cfg 文件

```
# sudo isocut -t /home/temp -k /etc/isocut/anaconda-ks.cfg -r /home/rpms
/home/isocut_iso/Kylin-Host-V10-General-Release-x86_64-Build05-20230906.iso
/home/result/new.iso
```

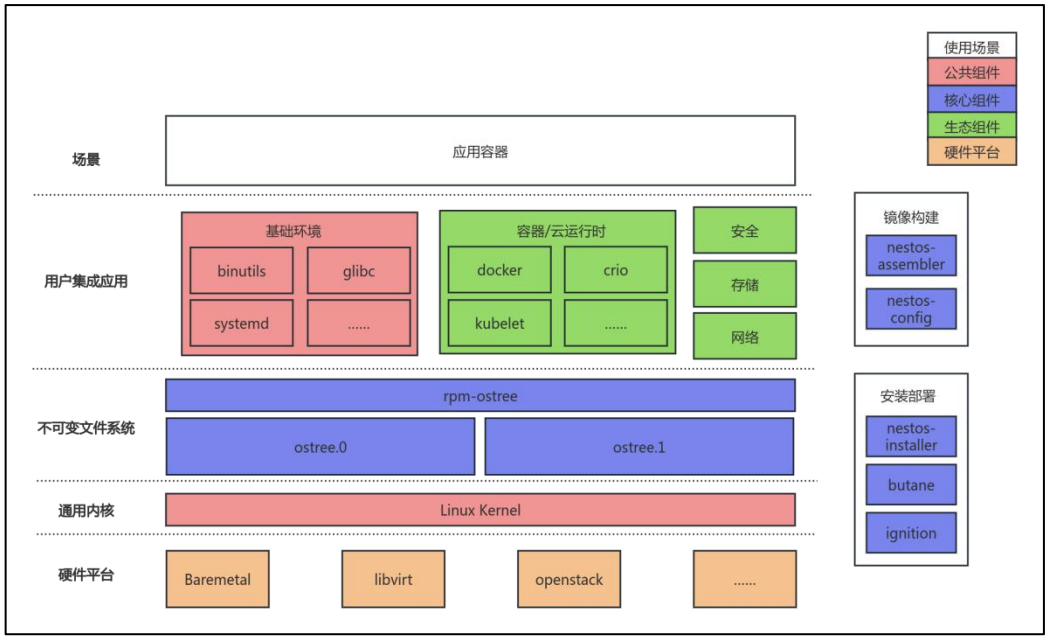
16. NestOS 不可变模式构建

16.1. NestOS 介绍

NestOS 是麒麟软件在 openEuler 社区开源孵化的云底座操作系统，集成了 rpm-ostree 支持、ignition 配置等技术，采用双根文件系统互为主备、原子化更新的设计思路，提供 nestos-assembler 工具快速集成构建。NestOS 针对 K8S、OpenStack 平台进行适配，优化容器运行底噪，使系统具备十分便捷的集群组建能力，可以更安全的运行大规模的容器化工作负载。

在银河麒麟云底座操作系统 V10 2406 中，我们提供基于 NestOS 技术路线的不可变操作系统镜像定制能力，以支持用户基于本产品软件源定制和使用不可变操作系统。本手册将完整阐述从构建、安装部署到使用 NestOS 的全流程，帮助用户充分利用 NestOS 的优势，快速高效地完成系统的配置和部署。

NestOS 适合作为以容器化应用为主的云场景基础运行环境，解决了在使用容器技术与容器编排技术实现业务发布、运维时与 底层环境高度解耦而带来的运维技术栈不统一，运维平台重复建设等问题，保证了业务与底座操作系统运维的一致性。



16.2. 环境准备

16.2.1. 构建环境要求

16.2.1.1. 制作构建工具 nestos-assembler 环境要求

- 推荐使用银河麒麟云底座操作系统 V10 2406 环境
- 内核需开启 ip 转发
- 存在/opt 目录且可写
- 剩余可用硬盘空间 > 5G

16.2.1.2. 构建 NestOS 环境要求

类别	要求
CPU	4vcpu
内存	4GB
硬盘	剩余可用空间>10GB
架构	x86_64 或 aarch64
其他	内核需开启 ip 转发； aarch64 环境需支持 kvm；

16.2.2. 部署配置要求

类别	推荐配置	最低配置
CPU	>4vcpu	1vcpu
内存	>4GB	512M
硬盘	>20GB	10GB
架构	x86_64、aarch64	

16.2.3. 其他约束

● 银河麒麟云底座操作系统 V10 2406 仅支持 NestOS 以容器镜像方式通过 rpm-ostree rebase 进行版本切换（升级），暂不支持 rpm-ostree deploy 相关指令。

● 在 NestOS 的构建和部署使用过程中，均需要开启 selinux。当前默认值为 permissive，用户可根据需求更改为 enforcing，但不可设置为 disabled。

16.3. 快速使用

16.3.1. 快速构建

在银河麒麟云底座操作系统 V10 2406 系统中，执行如下命令：

1）安装 epkl 软件源

```
dnf install -y kylin-epkl-repo
```

2）安装必要软件包

```
dnf install -y docker nestos-assembler nosa nestos-config
```

3）调整构建配置

nestos-config 软件包提供默认构建配置，无需额外操作。如需调整，请参考第 16.5 节。

4）制作 nestos-assembler 容器镜像

```
nosa-build -b <Base Image>
```

推荐使用基于 Kylin HostOS V10 的 base 镜像，更多说明请参考第 16.6.1 小节。

5）NestOS 镜像构建

```
# 在空目录中初始化构建环境
nosa init
# 拉取构建配置、更新缓存
nosa fetch
# 生成根文件系统、qcow2 及 OCI 镜像
nosa build
# 生成 live iso 及 PXE 镜像
nosa buildextend-metal
nosa buildextend-metal4k
nosa buildextend-live
```

详细构建及部署流程请参考第 16.6 节。

16.3.2. 快速部署

以 NestOS ISO 镜像为例，启动进入 live 环境后，执行如下命令根据向导提示完成安装：

```
sudo installnestos
```

其他部署方式请参考第 16.8 节。

16.4. 系统默认配置

选项	默认配置
docker 服务	默认 disable，需主动开启
selinux policy	默认 permissive
ssh 服务安全策略	默认仅支持密钥登录

16.5. 构建配置 nestos-config

16.5.1. 安装默认配置

- 开启 epkl 软件源

```
dnf install -y kylin-epkl-repo
```

- 安装默认构建配置软件包

```
dnf install -y nestos-config
```

注：安装时会将本地 repo 源文件拷贝至安装目录/opt/nestos-config 作为构建 nestos 的软件源，若您不期望这种行为，可以参考第 16.5.5.1 小节进行软件源的修改。

16.5.2. 配置目录结构说明

默认安装目录为/opt/nestos-config，以下是该目录介绍

目录/文件	说明
live/*	构建 liveiso 的引导配置
overlay.d/*	自定义文件配置
tests/*	用户自定义测试用例配置
*.repo	repo 源配置
.yaml，manifests/	主要构建配置

16.5.3. 主要文件解释

16.5.3.1. repo 文件

目录下的 repo 文件可用来配置用于构建 nestos 的软件仓库。

16.5.3.2. yaml 配置文件

目录下的 yaml 文件主要是提供 nestos 构建的各种配置，详见第 16.5.4 小节。

16.5.4. 主要字段解释

字段名称	作用
packages-aarch64、packages-x86_64、packages	软件包集成范围
exclude-packages	软件包集成黑名单
remove-from-packages	从指定软件包删除文件(夹)
remove-files	删除特定文件(夹)
extra-kargs	额外内核引导参数
initramfs-args	initramfs 构建参数
postprocess	文件系统构建后置脚本
default-target	配置 default-target，如 multi-user.target
rolij.name、releasever	镜像相关信息(镜像名称、版本)
lockfile-repos	构建可使用的仓库名列表,与 5.3.1 介绍的 repo 文件中的仓库名需要对应

16.5.5. 用户可配置项说明

16.5.5.1. repo 源配置

1. 在配置目录编辑 repo 文件，将内容修改为期望的软件仓库

```
$ cd /opt/nestos-config
$ vim template.repo
[repo_name_1]
baseurl = https://ip.address/1
enabled = 1

[repo_name_2]
baseurl = https://ip.address/2
enabled = 1
```

2. 修改 yaml 配置文件中的 lockfile-repo 字段内容为相应的仓库名称列表

注：仓库名称为 repo 文件中[]内的内容，不是 name 字段内容

```
$ cd /opt/nestos-config
$ vim manifests/rpmlist.yaml
修改 lockfile-repo 字段内容为
lockfile-repos:
- repo_name_1
- repo_name_2
```

注：配置的软件仓库源使用时需要注意网络连通性

16.5.5.2. 软件包裁剪

修改 packages、packages-aarch64、packages-x86_64 字段，可在其中添加或删除软件包，注意默认配置文件中 necessary 为最小列表不可删除。

如下所示，在 package 字段中添加了 bash-completion，构建安装后系统中会有 bash-completion 。

```
$ cd /opt/nestos-config
$ vim manifests/rpmlist.yaml
packages:
# necessary
- bootupd
...
# enhancement
- authselect
- bash-completion
...
packages-aarch64:
- grub2-efi-aa64
packages-x86_64:
- microcode_ctl
- grub2-efi-x64
```

16.5.5.3. 自定义镜像名称与版本号

修改 yaml 文件中的 releasever 及 rolij.name 字段，这些字段分别控制镜像的版本号及名称。

```
$ cd /opt/nestos-config
$ vim manifests/manifest.yaml

releasever: "1.0"
rojig:
  license: MulanPSL2 and MIT
  name: nestos
  summary: NestOS stable
```

如上配置，构建出的镜像格式为：

nestos-1.0.$(date "+%Y%m%d").$build_num.$type

其中 build_num 为构建次数，type 为类型后缀。

16.5.5.4. 自定义镜像中的 release 信息

正常 release 信息是由我们集成的 release 包（如 kylin-release）提供的，但是我们也可以通过添加 postprocess 脚本对/etc/os-release 文件进行重写操作。

```
$ cd /opt/nestos-config
$ vim manifests/workaround.yaml
在 postprocess 添加如下内容，若已存在相关内容，则只需修改对应 release 信息即可
postprocess:
 - |
   #!/usr/bin/env bash
   set -xeuo pipefail
   export OSTREE_VERSION="$(tail -1 /etc/os-release)"
   date_now=$(date "+%Y%m%d")
   echo -e 'NAME="Kylin HostOS V10 NestOS"\nVERSION="V10 (Nitrogen)"\nID="kylin"\nVERSION_ID="V10"\nPRETTY_NAME="Kylin HostOS V10 (Nitrogen) NestOS"\nANSI_COLOR="0;31"\nBUILDID="'${date_now}'"\nVARIANT="NestOS"\nVARIANT_ID="nestos"\n' > /usr/lib/os-release
   echo -e $OSTREE_VERSION >> /usr/lib/os-release
   cp -f /usr/lib/os-release /etc/os-release
```

使用如上配置进行镜像构建，启动构建出的镜像，查看系统中的 release 信息即为我们自定义内容。

```
[nest@nosa-devsh ~]$ cat /etc/os-release
NAME="Kylin HostOS V10 NestOS"
VERSION="V10 (Nitrogen)"
ID="kylin"
VERSION_ID="V10"
PRETTY_NAME="Kylin HostOS V10 (Nitrogen) NestOS"
ANSI_COLOR="0;31"
BUILDID="20240603"
VARIANT="NestOS"
VARIANT_ID="nestos"
```

16.5.5.5. 集成自定义文件

在 overlay.d 目录下每个目录进行自定义文件的添加和修改，这种操作可以实现构建镜像内容的自定义。

```
$ cd /opt/nestos-config
mkdir -p overlay.d/15nestos/etc/test
echo "This is a test message !" > overlay.d/15nestos/etc/test/test.txt
```

使用如上配置进行镜像构建，启动构建出的镜像，查看系统中对应文件内容即为我们上述自定义添加的内容。

```
$ cat /etc/test/test.txt
This is a test message !
```

16.6. 构建流程

NestOS 采用容器化方式，将构建工具链集成为一个完整的容器镜像，称为 NestOS-assembler。Kylin HostOS V10 提供构建 NestOS-assembler 容器镜像能力，方便用户使用。使用该容器镜像，用户可在任意 linux 发行版环境中构建多种形态 NestOS 镜像（例如在现有 CICD 环境中使用），也可对构建发布件进行管理、调试和自动化测试。

16.6.1. 制作构建工具 NestOS-assembler 容器镜像

16.6.1.1. 前置步骤

1) 准备容器 base 镜像

NestOS-assembler 容器镜像需要基于支持 yum/dnf 软件包管理器的 base 镜像制作，理论上可由任意发行版 base 镜像制作，但为最大程度减少软件包兼容性问题，仍推荐使用基于 Kylin HostOS V10 的 base 镜像。

2） 安装必要软件包

为方便用户快速构建，Kylin HostOS V10 以 rpm 包形式提供 nestos-assembler 容器镜像构建所需必要文件；提供 nosa-build 命令，简化封装构建过程，但需由用户指定 base 镜像。

主要步骤如下：

- 开启 epkl 软件源

```
dnf install -y kylin-epkl-repo
```

- 安装预编译软件包 nestos-assembler 与必备依赖 docker

```
dnf install -y nestos-assembler docker
```

3） 确认 docker 环境访问网络正常

根据运行环境网络情况，请确保 docker 环境访问网络正常，开启内核 ip 转发参数，必要时可重启 docker 服务重试。

16.6.1.2. 构建 NestOS-assembler 容器镜像

如您使用 Kylin HostOS V10 环境构建 nestos-assembler 容器镜像，执行以下命令指定 base 镜像即可：

```
nosa-build -b <Base Image>
```

nosa-build 命令调用 docker build 默认构建镜像名称为 nestos-assembler:latest，如您希望指定容器镜像名称或 tag，可执行如下命令：

```
nosa-build -b <Base Image> -t <name:tag>
```

默认情况下 nosa-build 使用当前环境中的 repo 信息，如您需要在非 Kylin HostOS V10 环境构建 nestos-assembler 容器镜像，需要在构建容器镜像时通过-r 参数指定 Kylin HostOS V10 的 repo 文件（注意添加 epkl 源）：

```
nosa-build -b <Base Image> -r /path/to/kylin_${basearch}.repo -r
/path/to/kylin-hostos-epkl.repo
```

您也可利用此功能添加自定义软件源。

如您需要在环境中指定 docker build 运行相关参数（例如构建阶段网络配置），nosa-build 命令提供构建参数透传，可在 nosa-build 命令后直接附上相关参数，nosa-build 会将所有未识别参数均传递至 docker build 阶段。

```
nosa-build -b <Base Image> [docker build options]
```

16.6.1.3. 其他说明

- nestos-assembler 软件包将构建容器镜像所需文件安装至/opt/nestos-assembler，该目录不得改名或移动位置，否则会导致 nosa-build 执行失败。
- nosa-build 命令每次执行会生成新的 Dockerfile 文件，具体路径为 /opt/nestos-assembler/Dockerfile-from_rpm 。
- nosa-build 命令‘-r’参数指定 repo 文件时，所有 repo 文件内容会被合并至 /opt/nestos-assembler/nestos-assembler.repo 供构建使用。
- nestos-assembler 软件包弱依赖 nosa 软件包，用于辅助使用 nestos-assembler 容器镜像，如制作 nestos-assembler 容器镜像与构建 NestOS 非同一环境，nosa 包可不安装。

16.6.2. 使用 NestOS-assembler 容器镜像

16.6.2.1. 前置步骤

1）准备 nestos-assembler 容器镜像

参考第 16.6.1 章节构建 nestos-assembler 容器镜像后，可通过私有化部署容器镜像仓库对该容器镜像进行管理和分发。请确保构建 NestOS 前，拉取适当版本的 nestos-assembler 容器镜像至当前环境。

2）安装辅助使用脚本 nosa

因 NestOS 构建过程需多次调用 nestos-assembler 容器镜像执行不同命令，同时需配置较多参数，为简化用户操作，提供 nosa 命令。如在非 Kylin HostOS V10 环境中构建 NestOS，同样建议安装该软件包/拷贝该脚本至构建环境。

```
dnf install -y nosa
```

16.6.2.2. 使用说明

（1）命令行参数说明

推荐通过 nosa 命令使用 nestos-assembler 容器镜像，具体支持参数如下：

- --image <nestos-assembler image>

指定 nestos-assembler 容器镜像名称（默认值为 nestos-assembler:latest）。

- --conf-dir <dir>

指定 NestOS 构建配置目录，参见第 16.5 节（默认值为/opt/nestos-config/）。

- --no-kvm

不使用 KVM 辅助虚拟化。NestOS 构建过程会在容器内部启动临时 qemu 虚机，如构建环境不支持 KVM，默认会构建失败。开启该选项后，qemu 采用 TCG 模式，可以正常构建 NestOS，但性能会严重降低。

- --add-certs

定制构建配置后，部分构建工作可能需要访问自签名证书保护的 URL 资源，此时需添加此选项将宿主机证书目录(/etc/pki/ca-trust) 挂载到构建容器中，此选项不支持挂载任意目录至容器内证书目录。

- -r, --arg <docker run args>

此选项可指定容器运行参数，即 docker run 支持的各项参数，可指定多个，但需使用双引号将完整的选项和参数作为此选项参数，例如：

```
nosa -r "-v local_path:container_path" -r "-m 2G"
```

- -h, --help

显示帮助信息，执行该命令时，除上述参数说明外，还会调用 nestos-assembler 容器镜像，输出支持的命令列表。当不存在 nestos-assembler 容器镜像或容器镜像不为默认值（nestos-assembler:latest）时，会提示如下信息：

```
The following commands are provided by the nestos-assembler container image:

Error: Print other commands failed.
Not have or set the correct nestos-assembler image?
```

此时可通过添加--image 参数输出完整帮助信息，示例如下：

```
nosa --image nestos-assembler:v1 --help
```

（2）环境变量说明

使用命令行参数简单直接，但多次调用时每次均需添加相应参数较为繁琐，为此，nosa 同样支持通过环境变量方式进行设置，整理支持的环境变量列表如下：

注意：环境变量方式优先级高于命令行参数方式，执行命令不符合预期时请首先检查是否存在遗留环境变量。

环境变量	说明
NESTOS_ASSEMBLER_CONTAINER	指定调用 nestos-assembler 容器镜像名称
NESTOS_ASSEMBLER_CONFIG_GIT	指定 NestOS 构建配置目录
NESTOS_ASSEMBLER_ADD_CERTS	存在任意值该环境变量时，挂载证书目录至容器
NESTOS_ASSEMBLER_CONTAINER_RUNTIME_ARGS	指定容器运行参数,即 docker run 支持的各项参数。特别地，如希望不使用 KVM 辅助虚拟化，可设置该环境变量值为"-e COSA_NO_KVM=1"

（3）构建工具命令一览

命令	功能说明
init	初始化构建环境及构建配置，详见 6.3
fetch	根据构建配置获取最新软件包至本地缓存
build	构建 ostree commit，是构建 NestOS 的核心命令
run	直接启动一个 qemu 实例，默认使用最新构建版本
prune	清理历史构建版本，默认保留最新 3 个版本
clean	删除全部构建发布件，添加--all 参数时同步清理本地缓存
list	列出当前构建环境中存在的版本及发布件
build-fast	基于前次构建记录快速构建新版本
push-container	推送容器镜像发布件至容器镜像仓库
buildextend-live	构建支持 live 环境的 ISO 发布件及 PXE 镜像
buildextend-metal	构建裸金属 raw 发布件
buildextend-metal4k	构建原生 4K 模式的裸金属 raw 发布件
buildextend-openstack	构建适用于 openstack 平台的 qcow2 发布件
buildextend-qemu	构建适用于 qemu 的 qcow2 发布件
basearch	获得当前架构信息
compress	压缩发布件
kola	自动化测试框架
kola-run	输出汇总结果的自动化测试封装
runc	以容器方式挂载当前构建根文件系统

tag	管理构建工程 tag
virt-install	通过 virt-install 为指定构建版本创建实例
meta	管理构建工程元数据
shell	进入 nestos-assembler 容器镜像

16.6.3. 准备构建环境

NestOS 构建环境需要独立的空文件夹作为工作目录，且支持多次构建，保留、管理历史构建版本。

创建构建环境前需首先准备构建配置（参考第 16.5 节）。

建议一份独立维护的构建配置对应一个独立的构建环境，即如果您希望构建多个不同用途的 NestOS，建议同时维护多份构建配置及对应的构建环境目录，这样可以保持不同用途的构建配置独立演进和较为清晰的版本管理。

16.6.3.1. 初始化构建环境

进入待初始化工作目录，执行如下命令即可以使用默认构建配置路径（/opt/nestos-config/）初始化构建环境：

```
nosa init
```

如需指定构建配置目录，可执行

```
nosa --conf-dir /path/to/nestos-config init
```

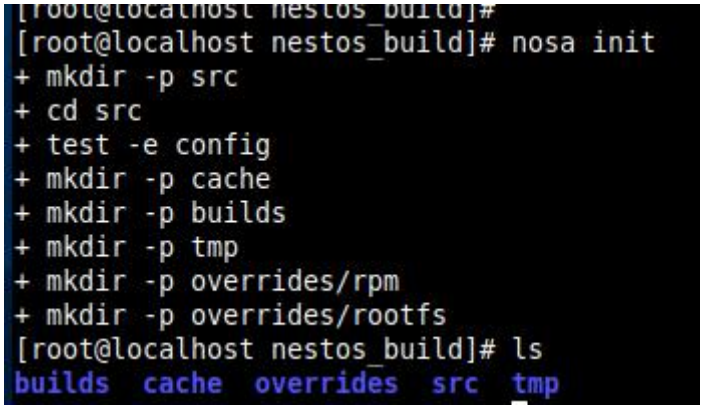
或

```
export NESTOS_ASSEMBLER_CONFIG_GIT=/path/to/nestos-config
nosa init
```

仅首次构建时需初始化构建环境，后续构建在不对构建配置做出重大更改的前提下，可重复使用该构建环境。

16.6.3.2. 构建环境说明

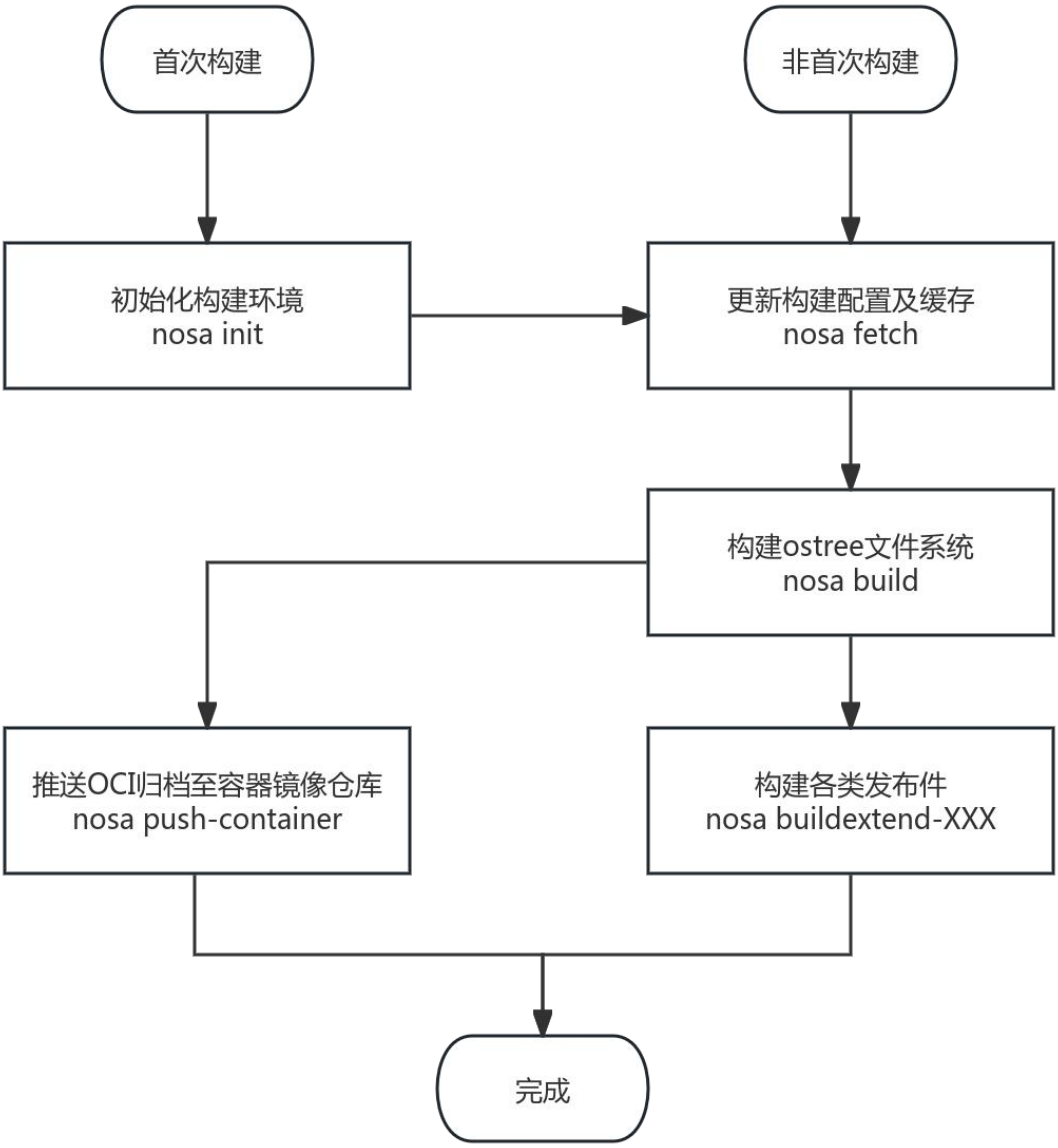
初始化完成后，工作目录创建出如下文件夹：



- **builds:**构建发布件及元数据存储目录，latest 子目录软链接指向最新构建版本。
- **cache:**缓存目录，根据构建配置中的软件源及软件包列表拉取至本地，历史构建 NestOS 的 ostree repo 均缓存于此目录。
- **overrides:**构建过程希望附加到最终发布件 rootfs 中的文件或 rpm 包可置于此目录。
- **src:**构建配置目录，指定本地目录时，由 docker 将指定目录挂载到此处，因此默认查看无实际文件，可通过 nosa shell 查看。
- **tmp:**临时目录，构建过程、自动化测试等场景均会使用该目录作为临时目录，构建发生异常时可在此处查看虚拟机命令行输出、journal 日志等信息。

16.6.4. 构建步骤

NestOS 构建主要流程及参考命令如下：



16.6.4.1. 初始化构建环境

首次构建时需初始化构建环境，详见第 16.6.3 小节。

非首次构建可直接使用原构建环境，可通过 nosa list 查看当前构建环境已存在版本及对应发布件。

16.6.4.2. 更新构建配置及缓存

初始化构建环境后，执行如下命令更新构建配置及缓存：

```
nosa fetch
```

该步骤初步校验构建配置是否可用，并通过配置的软件源拉取软件包至本地缓存。当构建配置发生变更或单纯希望更新软件源中最新版本软件包，均需要重新执行该步骤，否则可能导致构建失败或不符合预期。

当构建配置发生较大变更，希望清空本地缓存重新拉取时，需执行如下命令：

```
nosa clean --all
```

16.6.4.3. 构建不可变根文件系统

NestOS 不可变操作系统的核心是基于 ostree 技术的不可变根文件系统，执行如下步骤构建 ostree 文件系统：

```
nosa build
```

如您希望在构建 NestOS 时，添加自定义文件或 rpm 包，请在执行 build 命令前将相应文件放入构建环境 overrides 目录下 rootfs/或 rpm/文件夹。

正常情况下构建需要依赖 KVM 虚拟化的支持，如因某些特殊原因（无法开启嵌套虚拟化、异构构建等），可通过添加‘--no-kvm’参数解决，但会导致性能严重下降。

```
nosa --no-kvm build
```

build 命令默认会生成 ostree 文件系统和 OCI 归档文件，您也可以在执行命令时同步添加 qemu、metal、metal4k 中的一个或多个，同步构建发布件，等效于后续继续执行 buildextend-qemu、buildextend-metal 和 buildextend-metal4k 命令。

```
nosa build qemu metal metal4k
```

16.6.4.4. 构建各类发布件

build 命令执行完毕后，可继续执行 buildextend-XXX 命令用于构建各类型发布件，具体介绍如下：

- 构建 qcow2 镜像

```
nosa buildextend-qemu
```

- 构建带 live 环境的 ISO 镜像或 PXE 启动组件

```
nosa buildextend-live
```

- 构建适用于 openstack 环境的 qcow2 镜像

```
nosa buildextend-openstack
```

- 构建适用于容器镜像方式更新的容器镜像

执行 nosa build 命令构建 ostree 文件系统时，会同时生成 ociarchive 格式镜像，该格式镜像可通过执行如下命令推送到本地容器引擎存储后端或远程镜像仓库，无需执行其他构建步骤。

```
nosa push-container [container-image-name]
```

推送到本地容器引擎后端时，需通过--transport 指定推送协议（详见下文）。

推送到远程镜像仓库时，仓库地址需附加到推送容器镜像名称中，示例如下：

```
nosa push-container [registry.example.com/container-image-name]
```

除标识镜像 tag 外，容器镜像名称中不得出现其他":"。如在镜像名称中未检测到":"，该命令会自动生成{latest_build}-{arch}格式的 tag。示例如下：

```
nosa push-container container-image-name:1.0.20240603.0-x86_64
```

该命令支持以下可选参数：

- --authfile ：指定登录远程镜像仓库的鉴权文件
- --insecure：如远程镜像仓库采用自签名证书等场景，添加该参数可不校验 SSL/TLS 协议
- --transport：指定目标镜像推送协议，默认为 docker，具体支持项及说明如下：
  - containers-storage：推送至 podman、crio 等容器引擎本地存储目录
  - dir：推送至指定本地目录
  - docker：以 docker API 推送至私有或远端容器镜像仓库
  - docker-archive：等效于 docker save 导出归档文件，可供 docker load 使用
  - docker-daemon：推送至 docker 容器引擎本地存储目录

16.6.5. 获取发布件

构建完毕后，发布件均生成于构建环境中如下路径：

builds/{version}/{arch}/

如您仅关心最新构建版本或通过 CI/CD 调用，提供 latest 目录软链接至最新版本

目录，即：

builds/latest/{arch}/

为方便传输，您可以调用如下命令，压缩发布件体积：

```
nosa compress
```

压缩后原文件会被移除，会导致部分调试命令无法使用。

您也可以调用解压命令恢复原文件：

```
nosa uncompress
```

16.6.6. 构建环境维护

在构建 NestOS 环境前后，可能存在如下需求，可使用推荐的命令解决相应问题：

- 清理历史或无效构建版本，以释放磁盘空间

您可以通过以下命令清理历史版本构建：

```
nosa prune
```

也可删除当前构建环境中的全部发布件：

```
nosa clean
```

如构建配置更换过软件源或历史缓存无保留价值，可彻底清理当前构建环境缓存：

```
nosa clean --all
```

- 临时运行构建版本实例，用于调试或确认构建正确

```
nosa run
```

可通过--qemu-image 或--qemu-iso 指定启动镜像地址，其余参数请参考 nosa run --help 说明。

实例启动后，构建环境目录会被挂载至/var/mnt/workdir，可通过构建环境目录传入测试所需工具或文件用于调试。

- 运行自动化测试

```
nosa kola run
```

该命令会执行预设的测试用例，也可在其后追加测试用例名称，单独执行单条用例。

```
nosa kola testiso
```

该命令会执行 iso 或 pxe live 环境安装部署测试，可作为构建工程的冒烟测试。

- 调试验证构建工具（NestOS-assembler）

```
nosa shell
```

该命令可启动进入构建工具链容器的 shell 环境，您可以通过此命令验证构建工具链工作环境是否正常。

16.7. 部署配置

16.7.1. 前言

在开始部署 NestOS 之前，了解和准备必要的配置是至关重要的。NestOS 通过点火文件（ignition 文件）提供了一系列灵活的配置选项，可以通过 Butane 工具进行管理，方便用户进行自动化部署和环境设置。

在本章节中，将详细的介绍 Butane 工具的功能和使用方法，并根据不同场景提供配置示例。这些配置将帮助您快速启动和运行 NestOS，在满足应用需求的同时，确保系统的安全性和可靠性。此外，还会介绍如何自定义镜像，将点火文件预集成至镜像中，以满足特定应用场景的需求，从而实现高效的配置和部署 NestOS。

16.7.2. Butane 简介

Butane 是一个用于将人类可读的 YAML 配置文件转换为 NestOS 点火文件（Ignition 文件）的工具。Butane 工具简化了复杂配置的编写过程，允许用户以更易读的格式编写配置文件，然后将其转换为适合 NestOS 使用的 JSON 格式。

NestOS 对 Butane 进行了适配修改，新增 nestos 变体支持和配置规范版本 v1.0.0，对应的点火（ignition）配置规范为 v3.3.0，确保了配置的稳定性和兼容性。

16.7.3. Butane 使用

配置 HostOS 的 repo 源，执行以下指令安装：

开启 epkl 软件源

```
dnf install -y kylin-epkl-repo
```

安装 butane 软件包

```
dnf install butane
```

编辑 example.yaml 并执行以下指令将其转换为点火文件 example.ign，其中关于 yaml 文件的编写，将在后续展开：

```
butane example.yaml -o  example.ign -p
```

16.7.4. 支持的功能场景

以下配置示例（example.yaml）简述了 NestOS 主要支持的功能场景和进阶使用方法。

16.7.4.1. 设置用户和组并配置密码/密钥

```
variant: nestos
version: 1.0.0
passwd:
  users:
    - name: nest
      ssh_authorized_keys:
        - ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDHn2eh...
    - name: jlebon
      groups:
        - wheel
      ssh_authorized_keys:
        - ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQDC5QFS...
        - ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIIveEaMRW...
    - name: miabbott
      groups:
        - docker
        - wheel
      password_hash: $y$j9T$aUmgEDoFIDPhGxEe2FUjc/$C5A...
      ssh_authorized_keys:
        - ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAACAQDTey7R...
```

16.7.4.2. 文件操作——以配置网卡为例

```
variant: nestos
version: 1.0.0
storage:
  files:
    - path: /etc/NetworkManager/system-connections/ens2.nmconnection
      mode: 0600
      contents:
        inline: |
          [connection]
          id=ens2
          type=ethernet
          interface-name=ens2
          [ipv4]
          address1=10.10.10.10/24,10.10.10.1
          dns=8.8.8.8;
          dns-search=
          may-fail=false
          method=manual
```

16.7.4.3. 创建目录、文件、软连接并配置权限

```yaml
variant: nestos
version: 1.0.0
storage:
  directories:
  - path: /opt/tools
    overwrite: true
  files:
    - path: /var/helloworld
      overwrite: true
      contents:
        inline: Hello, world!
      mode: 0644
      user:
        name: dnsmasq
      group:
        name: dnsmasq
    - path: /opt/tools/transmogrifier
      overwrite: true
      contents:
        source: https://mytools.example.com/path/to/archive.gz
        compression: gzip
        verification:
          hash:
sha512-00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
      mode: 0555
  links:
    - path: /usr/local/bin/transmogrifier
      overwrite: true
      target: /opt/tools/transmogrifier
      hard: false
```

16.7.4.4. 编写 systemd 服务——以启停容器为例

```
variant: nestos
version: 1.0.0
systemd:
  units:
    - name: hello.service
      enabled: true
      contents: |
        [Unit]
        Description=MyApp
        After=network-online.target
        Wants=network-online.target

        [Service]
        TimeoutStartSec=0
        ExecStartPre=-/bin/podman kill busybox1
        ExecStartPre=-/bin/podman rm busybox1
        ExecStartPre=/bin/podman pull busybox
        ExecStart=/bin/podman run --name busybox1 busybox /bin/sh -c
""trap 'exit 0' INT TERM; while true; do echo Hello World; sleep 1; done""

        [Install]
        WantedBy=multi-user.target
```

16.7.5. 点火文件预集成

NestOS 构建工具链支持用户根据实际使用场景和需求定制镜像。在镜像制作完成后，nestos-installer 还提供了针对镜像部署与应用等方面进行自定义的一系列功能，如嵌入点火文件、预分配安装位置、增删内核参数等功能，以下将针对主要功能进行介绍。

16.7.5.1. 点火文件预集成至 ISO 镜像

要将点火文件预集成至 ISO 镜像中，我们需要准备好 NestOS 的 ISO 镜像，环境已安装 nestos-installer 软件包；编写部署配置文件，并使用 butane 工具将其转换为 ign 文件。示例文件中，我们仅配置简单的用户名和密码（密码要求加密，示例中为 qwer1234），内容如下：

```
variant: nestos
version: 1.0.0
passwd:
  users:
    - name: root
      password_hash: "$1$root$CPjzNGH.NqmQ7rh26EeXv1"
```

将上述 yaml 转换为 ign 文件后，执行如下指令嵌入点火文件并指定目标磁盘位置，其中 xxx.iso 为准备至本地的 NestOS ISO 镜像：

```
nestos-installer iso customize --dest-device /dev/sda --dest-ignition
example.ign xxx.iso
```

使用该集成点火文件的 ISO 镜像进行安装时，NestOS 会自动读取点火文件并安装至目标磁盘，待进度条完成度为 100%后，自动进入安装好的 NestOS 环境，用户可根据 ign 文件配置的用户名和密码进入系统。

16.7.5.2. 点火文件预集成至 PXE 镜像

准备好 NestOS 的 PXE 镜像至本地，组件获取方式参考第 16.6.5 小节，其他步骤同上。

为了方便用户使用，nestos-installer 也支持从 ISO 镜像中提取 PXE 组件的功能，执行如下指令，其中 xxx.iso 为保存至本地的 NestOS ISO 镜像：

```
nestos-installer iso extract pxe xxx.iso
```

得到如下输出件：

```
xxx-initrd.img
xxx-rootfs.img
xxx-vmlinuz
```

执行如下指令嵌入点火文件并指定目标磁盘位置：

```
nestos-installer pxe customize --dest-device /dev/sda --dest-ignition
example.ign xxx-initrd.img --output  custom-initrd.img
```

根据使用 PXE 安装 NestOS 的方式，替换相应的 xxx-initrd.img 为 custom-initrd.img。启动后 NestOS 会自动读取点火文件并安装至目标磁盘，待进度条完成度为 100%后，自动进入安装好的 NestOS 环境，用户可根据 ign 文件配置的用户名和密码进入系统。

16.8. 部署流程

16.8.1. 简介

NestOS 支持多种部署平台及常见部署方式，当前主要支持 qcow2、ISO 与 PXE 三种部署方式。与常见通用 OS 部署相比，主要区别在于如何传入以 ign 文件为特征的自定义部署配置，以下各部分将会分别介绍。

16.8.2. 使用 qcow2 镜像安装

16.8.2.1. 使用 qemu 创建 qcow2 实例

准备 NestOS 的 qcow2 镜像及相应点火文件（详见第 18.7 节），终端执行如下步骤：

```
IGNITION_CONFIG="/path/to/example.ign"
IMAGE="/path/to/image.qcow2"
IGNITION_DEVICE_ARG="-fw_cfg
name=opt/com.coreos/config,file=${IGNITION_CONFIG}"

qemu-img create -f qcow2 -F qcow2 -b ${IMAGE} my-nestos-vm.qcow2
```

aarch64 环境执行如下命令：

```
qemu-kvm -m 2048 -M virt -cpu host -nographic -drive
if=virtio,file=my-nestos-vm.qcow2 ${IGNITION_DEVICE_ARG} -nic
user,model=virtio,hostfwd=tcp::2222-:22 -bios
/usr/share/edk2/aarch64/QEMU_EFI-pflash.raw
```

x86_64 环境执行如下命令：

```
qemu-kvm -m 2048 -M pc -cpu host -nographic -drive
if=virtio,file=my-nestos-vm.qcow2 ${IGNITION_DEVICE_ARG} -nic
user,model=virtio,hostfwd=tcp::2222-:22
```

16.8.2.2. 使用 virt-install 创建 qcow2 实例

假设 libvirt 服务正常，网络默认采用 default 子网，绑定 virbr0 网桥，您可参考以下步骤创建 NestOS 实例。

准备 NestOS 的 qcow2 镜像及相应点火文件（详见第 7 章），终端执行如下步骤：

```
IGNITION_CONFIG="/path/to/example.ign"
IMAGE="/path/to/image.qcow2"
VM_NAME="nestos"
VCPUS="4"
RAM_MB="4096"
DISK_GB="10"
IGNITION_DEVICE_ARG=(--qemu-commandline="-fw_cfg
name=opt/com.coreos/config,file=${IGNITION_CONFIG}")
```

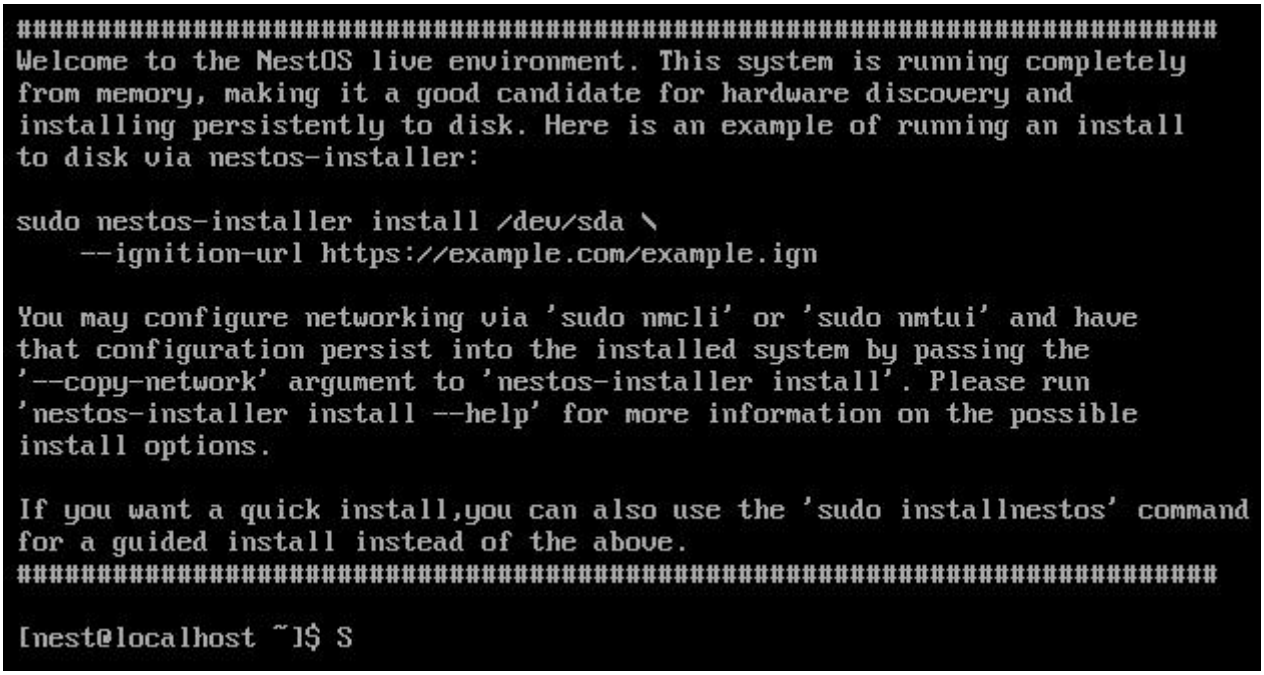
注意：使用 virt-install 安装，qcow2 镜像及 ign 文件需指定绝对路径。

执行如下命令创建实例：

```
virt-install --connect="qemu:///system" --name="${VM_NAME}"
--vcpus="${VCPUS}" --memory="${RAM_MB}"
--os-variant="kylin-hostos10.0" --import --graphics=none
--disk="size=${DISK_GB},backing_store=${IMAGE}"  --network
bridge=virbr0 "${IGNITION_DEVICE_ARG[@]}"
```

16.8.3. 使用 ISO 镜像安装

准备 NestOS 的 ISO 镜像并启动。首次启动的 NestOS ISO 镜像会默认进入 Live 环境，该环境为易失的内存环境。



16.8.3.1. 通过 nestos-installer 安装向导脚本安装 OS 至目标磁盘

1）在 NestOS 的 Live 环境中，根据首次进入的打印提示，可输入以下指令，即可自动生成一份简易的点火文件并自动安装重启

```
sudo installnestos
```

2）根据终端提示信息依次输入用户名和密码；

3）选择目标磁盘安装位置，可直接选择回车设置为默认项/dev/sda；

4）执行完以上步骤后，nestos-installer 开始根据我们提供的配置将 NestOS 安装至目标磁盘，待进度条 100%后，自动重启；

5）重启后自动进入 NestOS，在 grub 菜单直接回车或者等待 5s 后启动系统，随后根据此前配置的用户名和密码进入系统。至此，安装完成。

16.8.3.2. 通过 nestos-installer 命令手动安装 OS 至目标磁盘

1）准备好点火文件 example.ign（详见第 16.7 节）；

2）根据首次进入 NestOS 的 Live 环境打印的提示，输入以下指令开始安装：

```
sudo nestos-installer install /dev/sda --ignition-file example.ign
```

如具备网络条件，点火文件也可通过网络获取，如：

```
sudo nestos-installer install /dev/sda --ignition-file
http://www.example.com/example.ign
```

3）执行完上述指令后，nestos-installer 开始根据我们提供的配置将 NestOS 安装至目标磁盘，待进度条 100%后，执行如下命令重新启动：

```
sudo systemctl reboot
```

4）重启后自动进入 NestOS，在 grub 菜单直接回车或者等待 5s 后启动系统，随后根据此前配置的用户名和密码进入系统。至此，安装完成

16.8.4. PXE 部署

NestOS 的 PXE 安装组件包括 kernel、initramfs.img 和 rootfs.img。这些组件以 nosa buildextend-live 命令生成（详见第 16.6 节）。

1）使用 PXELINUX 的 kernel 命令行指定内核，简单示例如下：

```
KERNEL nestos-live-kernel-x86_64
```

2）使用 PXELINUX 的 append 命令行指定 initrd 和 rootfs，简单示例如下：

```
APPEND
initrd=nestos-live-initramfs.x86_64.img,nestos-live-rootfs.x86_64.img
```

注意：如您采用 16.7.5 小节所述，已将点火文件预集成至 PXE 组件，则仅需在此进行替换，无需执行后续步骤。

3）指定安装位置，以/dev/sda 为例，在 APPEND 后追加，示例如下：

```
nestosos.inst.install_dev=/dev/sda
```

4）指定点火文件，需通过网络获取，在 APPEND 后追加相应地址，示例如下：

```
nestos.inst.ignition_url=http://www.example.com/example.ign
```

5）启动后 NestOS 会自动读取点火文件并安装至目标磁盘，待进度条完成度为100%后，自动进入安装好的 NestOS 环境，用户可根据 ign 文件配置的用户名和密码进入系统。

16.9. 基本使用

16.9.1. 简介

NestOS 采用基于 ostree 和 rpm-ostree 技术的操作系统封装方案，将关键目录设置为只读状态，核心系统文件和配置不会被意外修改；采用 overlay 分层思想，允许用户在基础 ostree 文件系统之上分层管理 RPM 包，不会破坏初始系统体系结构；同时支持构建 OCI 格式镜像，实现以镜像为最小粒度进行操作系统版本的切换。

16.9.2. SSH 连接

出于安全考虑，NestOS 默认不支持用户使用密码进行 SSH 登录，而只能使用密钥认证方式。这一设计旨在增强系统的安全性，防止因密码泄露或弱密码攻击导致的潜在安全风险。

NestOS 通过密钥进行 SSH 连接的方法与 Kylin HostOS V10 一致，如果用户需要临时开启密码登录，可按照以下步骤执行：

1）编辑 ssh 服务附加配置文件

```
vi /etc/ssh/sshd_config.d/40-disable-passwords.conf
```

2）修改默认配置 PasswordAuthentication 为如下内容：

```
PasswordAuthentication yes
```

3）重启 sshd 服务，便可实现临时使用密码进行 SSH 登录。

16.9.3. RPM 包安装

注意：不可变操作系统不提倡在运行环境中安装软件包，提供此方法仅供临时调试等场景使用，因业务需求需要变更集成软件包列表请通过更新构建配置重新构建实现。

NestOS 不支持常规的包管理器 dnf/yum，而是通过 rpm-ostree 来管理系统更新和软件包安装。rpm-ostree 结合了镜像和包管理的优势，允许用户在基础系统之上分层安装和管理 rpm 包，并且不会破坏初始系统的结构。使用以下命令安装 rpm 包：

```
rpm-ostree install <packagename>
```

安装完成后，重新启动操作系统，可以看到引导加载菜单出现了新分支，默认第一个分支为最新的分支

```
systemctl reboot
```

重启进入系统，查看系统包分层状态，可看到当前版本已安装<packagename>

```
rpm-ostree status -v
```

16.9.4. 版本回退（临时/永久）

更新/rpm 包安装完成后，上一版本的操作系统部署仍会保留在磁盘上。如果更新导致问题，用户可以使用 rpm-ostree 进行版本回退，这一步操作需要用户手动操作，具体流程如下：

16.9.4.1. 临时回退

要临时回滚到之前的 OS 部署，在系统启动过程中按住 shift 键，当引导加载菜单出现时，在菜单中选择相应的分支（默认有两个，选择另外一个即可）。在此之前，可以使用以下指令查看当前环境中已存在的两个版本分支：

```
rpm-ostree status
```

16.9.4.2. 永久回退

要永久回滚到之前的操作系统部署，用户需在当前版本中运行如下指令，此操作将使用之前版本的系统部署作为默认部署。

```
rpm-ostree rollback
```

重新启动以生效，引导加载菜单的默认部署选项已经改变，无需用户手动切换。

```
systemctl reboot
```

16.10. 容器镜像方式更新

16.10.1. 应用场景说明

NestOS 作为基于不可变基础设施思想的容器云底座操作系统，将文件系统作为一个整体进行分发和更新。这一方案在运维与安全方面带来了巨大的便利。然而，在实际生产环境中，官方发布的版本往往难以满足用户的需求。例如，用户可能希望在系统中默认集成自维护的关键基础组件，或者根据特定场景的需求对软件包进行进一步的裁剪，以减少系统的运行负担。因此，与通用操作系统相比，用户对 NestOS 有着更强烈和更频繁的定制需求。

NestOS-assembler 可提供符合 OCI 标准的容器镜像，且不仅是将根文件系统打包分发，利用 ostree native container 特性，可使容器云场景用户使用熟悉的技术栈，只需编写一个 ContainerFile(Dockerfile)文件，即可轻松构建定制版镜像，用于自定义集成组件或后续的升级维护工作。

16.10.2. 使用方式

16.10.2.1. 定制镜像

- 基本步骤

（1）参考第 16.6 节构建 NestOS 容器镜像，可使用 nosa push-container 命令推送至公共或私有容器镜像仓库。

（2）编写 Containerfile(Dockerfile)示例如下：

```
FROM registry.example.com/nestos:1.0.20240603.0-x86_64

# 执行自定义构建步骤，例如安装软件或拷贝自构建组件
# 此处以安装 strace 软件包为例
RUN rpm-ostree install strace && rm -rf /var/cache && ostree container commit
```

（3）执行 docker build 或集成于 CICD 中构建相应镜像

- 注意事项

(1) NestOS 无 yum/dnf 包管理器，如需安装软件包可采用 rpm-ostree install 命令安装本地 rpm 包或软件源中提供软件

(2) 如有需求也可修改/etc/yum.repo.d/目录下软件源配置

(3) 每层有意义的构建命令末尾均需添加&& ostree container commit 命令，从构建容器镜像最佳实践角度出发，建议尽可能减少 RUN 层的数量

(4) 构建过程中会对非/usr 或/etc 目录内容进行清理，因此通过容器镜像方式定制主要适用于软件包或组件更新，请勿通过此方式进行系统维护或配置变更（例如添加用户 useradd）

16.10.2.2. 部署/升级镜像

假设上述步骤构建容器镜像被推送为 registry.example.com/nestos:1.0.20240603.0-x86_64。

在已部署 NestOS 的环境中执行如下命令：

```
sudo rpm-ostree rebase
ostree-unverified-registry:registry.example.com/nestos:1.0.20240603.0-x86_64
```

重新引导后完成定制版本部署。

当您使用容器镜像方式部署后，rpm-ostree upgrade 默认会将版本发布流从 ostree 发布流地址切换为容器镜像地址。之后，您可以在相同的 tag 下更新容器镜像，使用 rpm-ostree upgrade 可以检测远端镜像是否已经更新，如果有变更，它会拉取最新的镜像并完成部署。

17. 附录 1：鲲鹏 BoostKit 机密计算 TrustZone 套件

参考鲲鹏官方文档检查鲲鹏服务器是否已预置 TrustZone 套件，详见以下链接：

https://www.hikunpeng.com/document/detail/zh/kunpengcctrustzone/fg-tz/kunpengtrustzone_20_0018.html