



Red Hat Enterprise Linux 8

管理、监控和更新内核

在 Red Hat Enterprise Linux 8 中管理 Linux 内核的指南

Red Hat Enterprise Linux 8 管理、监控和更新内核

在 Red Hat Enterprise Linux 8 中管理 Linux 内核的指南

法律通告

Copyright © 2024 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux[®] is the registered trademark of Linus Torvalds in the United States and other countries.

Java[®] is a registered trademark of Oracle and/or its affiliates.

XFS[®] is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL[®] is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js[®] is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack[®] Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

摘要

作为系统管理员，您可以配置 Linux 内核以优化操作系统。对 Linux 内核进行修改可以提高系统性能、安全性和稳定性，并可以对系统进行审核并对问题进行故障排除。

目录

对红帽文档提供反馈	7
第 1 章 LINUX 内核	8
1.1. 内核是什么	8
1.2. RPM 软件包	8
1.3. LINUX 内核 RPM 软件包概述	8
1.4. 显示内核软件包的内容	9
1.5. 安装特定的内核版本	10
1.6. 更新内核	10
1.7. 将内核设置为默认	11
第 2 章 管理内核模块	13
2.1. 内核模块简介	13
2.2. 内核模块依赖关系	13
2.3. 列出已安装的内核模块	14
2.4. 列出当前载入的内核模块	14
2.5. 列出所有安装的内核	15
2.6. 显示内核模块信息	15
2.7. 在系统运行时载入内核模块	16
2.8. 在系统运行时卸载内核模块	17
2.9. 在启动过程早期卸载内核模块	18
2.10. 在系统引导时自动载入内核模块	19
2.11. 防止在系统引导时自动载入内核模块	20
2.12. 编译自定义的内核模块	22
第 3 章 为安全引导签名内核和模块	25
3.1. 先决条件	25
3.2. 什么是 UEFI 安全引导	26
3.3. UEFI 安全引导支持	26
3.4. 使用 X.509 密钥验证内核模块的要求	27
3.5. 公钥的源	28
3.6. 生成公钥和私钥对	29
3.7. 系统密钥环输出示例	30
3.8. 通过在 MOK 列表中添加公钥在目标系统中注册公钥	31
3.9. 使用私钥签名内核	32
3.10. 使用私钥签名 GRUB 构建	33
3.11. 使用私钥签名内核模块	34
3.12. 载入经过签名的内核模块	36
第 4 章 配置内核命令行参数	38
4.1. 什么是内核命令行参数	38
4.2. 了解引导条目	38
4.3. 为所有引导条目更改内核命令行参数	39
4.4. 为单一引导条目更改内核命令行参数	40
4.5. 在引导时临时更改内核命令行参数	41
4.6. 配置 GRUB 设置以启用串行控制台连接	42
第 5 章 在运行时配置内核参数	43
5.1. 什么是内核参数	43
5.2. 使用 SYSCTL 临时配置内核参数	44
5.3. 使用 SYSCTL 永久配置内核参数	44
5.4. 使用 /ETC/SYSCTL.D/ 中的配置文件调整内核参数	45

5.5. 通过 /PROC/SYS/ 临时配置内核参数	45
5.6. 其它资源	46
第 6 章 对 GRUB 菜单进行临时更改	47
6.1. GRUB 简介	47
6.2. 引导装载程序规格介绍	48
6.3. 引导至救援模式	48
6.4. 引导至紧急模式	49
6.5. 引导至 DEBUG SHELL	50
6.6. 连接到 DEBUG SHELL	50
6.7. 使用安装盘重置 ROOT 密码	51
6.8. 使用 RD.BREAK 重置 ROOT 密码	53
6.9. 其它资源	55
第 7 章 对 GRUB 引导装载程序进行永久性更改	56
7.1. 先决条件	56
7.2. 列出默认的内核	56
7.3. 查看内核的 GRUB 菜单条目	56
7.4. 编辑内核参数	57
7.5. 在 GRUB 菜单条目中添加和删除参数	57
7.6. 添加一个新的引导条目	58
7.7. 使用 GRUBBY 更改默认引导条目	59
7.8. 使用同样的参数更新所有内核菜单	59
7.9. 为当前和将来的内核更改默认内核选项	59
7.10. 其它资源	60
第 8 章 构建一个自定义引导菜单	61
8.1. GRUB 配置文件	61
8.2. 隐藏可引导的内核的列表	61
8.3. 使用 GRUB 配置文件更改默认引导条目	62
第 9 章 重新安装 GRUB	63
9.1. 在基于 BIOS 的机器上重新安装 GRUB	63
9.2. 在基于 UEFI 的机器上重新安装 GRUB	63
9.3. 在 IBM POWER 机器上重新安装 GRUB	64
9.4. 重置 GRUB	64
第 10 章 使用密码保护 GRUB	66
10.1. 仅为修改菜单条目设置密码保护	66
10.2. 对修改和引导菜单条目设置密码保护	66
第 11 章 在虚拟环境中保留内核 PANIC 参数	68
11.1. 什么是软锁定	68
11.2. 控制内核 PANIC 的参数	68
11.3. 在虚拟环境中伪装软锁定	69
第 12 章 为数据库服务器调整内核参数	70
12.1. 介绍	70
12.2. 影响数据库应用程序性能的参数	70
第 13 章 内核日志记录入门	72
13.1. 什么是内核环缓冲	72
13.2. 日志级别和内核日志记录上的 PRINTK 角色	72
第 14 章 安装 KDUMP	74
14.1. 什么是 KDUMP	74

14.2. 使用 ANACONDA 安装 KDUMP	74
14.3. 在命令行上安装 KDUMP	75
第 15 章 在命令行中配置 KDUMP	76
15.1. 估算 KDUMP 大小	76
15.2. 配置 KDUMP 内存用量	76
15.3. 配置 KDUMP 目标	77
15.4. 配置 KDUMP 核心收集器	80
15.5. 配置 KDUMP 默认失败响应	81
15.6. KDUMP 的配置文件	82
15.7. 测试 KDUMP 配置	83
15.8. 系统崩溃后 KDUMP 产生的文件	84
15.9. 启用和禁用 KDUMP 服务	84
15.10. 防止内核驱动程序为 KDUMP 加载	85
15.11. 在使用加密磁盘的系统中运行 KDUMP	86
第 16 章 启用 KDUMP	88
16.1. 为所有安装的内核启用 KDUMP	88
16.2. 为特定安装的内核启用 KDUMP	88
16.3. 禁用 KDUMP 服务	89
第 17 章 在 WEB 控制台中配置 KDUMP	91
17.1. 在 WEB 控制台中配置 KDUMP 内存用量和目标位置	91
第 18 章 支持的 KDUMP 配置和目标	94
18.1. KDUMP 的内存要求	94
18.2. 自动内存保留的最小阈值	95
18.3. 支持的 KDUMP 目标	95
18.4. 支持的 KDUMP 过滤等级	97
18.5. 支持的默认故障响应	98
18.6. 使用 FINAL_ACTION 参数	98
18.7. 使用 FAILURE_ACTION 参数	99
第 19 章 固件支持的转储机制	100
19.1. IBM POWERPC 硬件支持转储固件	100
19.2. 启用固件支持的转储机制	100
19.3. IBM Z 硬件支持的固件转储机制	101
19.4. 在 FUJITSU PRIMEQUEST 系统中使用 SADUMP	101
第 20 章 分析内核转储	103
20.1. 安装 CRASH 工具	103
20.2. 运行和退出 CRASH 工具	103
20.3. 在 CRASH 工具中显示各种指示符	104
20.4. 使用 KERNEL OOPS ANALYZER	107
20.5. KDUMP HELPER 工具	108
第 21 章 使用早期 KDUMP 来捕获引导时间崩溃	109
21.1. 什么是早期 KDUMP	109
21.2. 启用早期 KDUMP	109
第 22 章 使用内核实时修补程序应用补丁	111
22.1. KPATCH 的限制	111
22.2. 对第三方实时补丁的支持	111
22.3. 获得内核实时补丁	112
22.4. 组件内核实时修补	112

22.5. 内核实时补丁如何工作	112
22.6. 将当前安装的内核订阅到实时补丁流	113
22.7. 自动订阅将来的内核到实时补丁流	114
22.8. 禁用向实时补丁流的自动订阅	116
22.9. 更新内核补丁模块	117
22.10. 删除 LIVE PATCHING 软件包	117
22.11. 卸载内核补丁模块	118
22.12. 禁用 KPATCH.SERVICE	120
第 23 章 使用控制组群为应用程序设置系统资源限制	122
23.1. 控制组简介	122
23.2. 内核资源控制器简介	123
23.3. 命名空间简介	124
23.4. 使用 CGROUPS-V1 为应用程序设置 CPU 限制	125
第 24 章 使用 CGROUPS-V2 控制应用程序的 CPU 时间分布	129
24.1. 挂载 CGROUPS-V2	129
24.2. 为 CPU 时间分布准备 CGROUP	130
24.3. 通过调整 CPU 带宽来控制应用程序的 CPU 时间分布	133
24.4. 通过调整 CPU 权重来控制应用程序的 CPU 时间分布	135
第 25 章 使用带有 SYSTEMD 的控制组群版本 1	138
25.1. 控制组群版本 1 中的 SYSTEMD 角色	138
25.2. 创建临时控制组群	139
25.3. 创建持久的控制组群	140
25.4. 在命令行中配置内存资源控制设置	140
25.5. 使用单元文件配置内存资源控制设置	141
25.6. 删除临时控制组群	142
25.7. 删除持久的控制组群	143
25.8. 列出 SYSTEMD 单元	143
25.9. 查看 SYSTEMD CGROUPS 层次结构	145
25.10. 查看资源控制器	146
25.11. 监控资源消耗	147
第 26 章 使用 CGROUPS-V2 和 SYSTEMD 配置资源管理	148
26.1. 先决条件	148
26.2. 资源分配模型简介	148
26.3. 使用 SYSTEMD 分配 CPU 资源	149
26.4. SYSTEMD 的 CPU 时间分配策略选项	149
26.5. 使用 SYSTEMD 分配内存资源	150
26.6. SYSTEMD 的内存分配配置选项	150
26.7. 使用 SYSTEMD 配置 I/O 带宽	151
26.8. SYSTEMD 的 I/O 带宽配置选项	152
26.9. 使用 SYSTEMD 配置 CPuset 控制器	153
第 27 章 使用 SYSTEMD 配置 CPU 关联性和 NUMA 策略	154
27.1. 使用 SYSTEMD 配置 CPU 关联性	154
27.2. 使用 SYSTEMD 配置 NUMA 策略	154
27.3. SYSTEMD 的 NUMA 策略配置选项	155
第 28 章 使用 BPF COMPILER COLLECTION 分析系统性能	157
28.1. 安装 BCC-TOOLS 软件包	157
28.2. 使用所选 BCC-TOOLS 进行性能调整	157
第 29 章 使用内核完整性子系统提高安全性	162

29.1. 内核完整性子系统	162
29.2. 可信和加密的密钥	162
29.3. 使用可信密钥	163
29.4. 使用加密密钥	165
29.5. 启用 IMA 和 EVM	166
29.6. 使用完整性测量架构收集文件哈希	169
第 30 章 使用 RHEL 系统角色永久配置内核参数	171
30.1. KERNEL_SETTINGS RHEL 系统角色简介	171
30.2. 使用 KERNEL_SETTINGS RHEL 系统角色应用所选内核参数	171
第 31 章 使用高级错误报告	174
31.1. AER 概述	174
31.2. 收集并显示 AER 消息	174

对红帽文档提供反馈

我们感谢您对我们文档的反馈。让我们了解如何改进它。

通过 Jira 提交反馈（需要帐户）

1. 登录到 [Jira](#) 网站。
2. 单击顶部导航栏中的 **Create**。
3. 在 **Summary** 字段中输入描述性标题。
4. 在 **Description** 字段中输入您的改进建议。包括到文档相关部分的链接。
5. 点对话框底部的 **Create**。

第 1 章 LINUX 内核

了解由红帽（红帽内核）提供和维护的 Linux 内核和 Linux 内核 RPM 软件包。使红帽内核保持更新，这可以确保操作系统具有最新的程序错误修复、性能增强和补丁，并与新硬件兼容。

1.1. 内核是什么

内核是 Linux 操作系统的核心部分，其管理系统资源，并提供硬件和软件应用程序之间的接口。

红帽内核是一个基于上游 Linux 主线内核的定制内核，红帽工程师对其进行了进一步的开发和强化，专注于稳定性和与最新技术和硬件的兼容性。

在红帽发布新内核版本前，内核需要通过一组严格的质量保证测试。

红帽内核以 RPM 格式打包，以便它们可以通过 YUM 软件包管理器轻松地升级和验证。



警告

红帽不支持不是由红帽编译的内核。

1.2. RPM 软件包

RPM 软件包由文件的存档和用来安装和擦除这些文件的元数据组成。具体来说，RPM 软件包包含以下部分：

GPG 签名

GPG 签名用于验证软件包的完整性。

标头（软件包元数据）

RPM 软件包管理器使用此元数据确定软件包依赖项、安装文件的位置以及其他信息。

payload

有效负载是一个 **cpio** 归档，其包含要安装到系统的文件。

RPM 软件包有两种类型。这两种类型都共享文件格式和工具，但内容不同，并实现不同的目的：

- 源 RPM (SRPM)
SRPM 包含源代码和 **spec** 文件，该文件描述了如何将源代码构建为二进制 RPM。另外，SRPM 可以包含源代码的补丁。

二进制 RPM

一个二进制 RPM 包含了根据源代码和补丁构建的二进制文件。

1.3. LINUX 内核 RPM 软件包概述

kernel RPM 是一个元数据软件包，它不包含任何文件，而是保证正确安装了以下子软件包：

kernel-core

包含内核的二进制镜像，所有用于引导系统的与 **initramfs** 相关的对象，以及用于确保核心功能的最小内核模块数。仅在虚拟和云环境中使用这个子软件包来为 Red Hat Enterprise Linux 8 内核提供一个快速引导时间和小磁盘空间。

kernel-modules

包含 **kernel-core** 中不存在的其余内核模块。

上述 **kernel** 子软件包中的一部分旨在帮助系统管理员减少需要维护的范围，特别是在虚拟化和云环境中。

例如，可选内核软件包：

kernel-modules-extra

包含用于罕见硬件的内核模块，以及默认加载被禁用的模块。

kernel-debug

包含一个为内核诊断启用了大量调试选项的内核，但代价是降低了性能。

kernel-tools

包含用于操作 Linux 内核和支持文档的工具。

kernel-devel

包含足以针对 **kernel** 软件包构建模块的内核标头和 makefile。

kernel-abi-stablelists

包含与 RHEL 内核 ABI 有关的信息，包括外部 Linux 内核模块所需的内核符号的列表以及帮助实施的 **yum** 插件。

kernel-headers

包括指定 Linux 内核和用户空间库以及程序间接口的 C 标头文件。头文件定义了构建大多数标准程序所需的常量结构和常量。

其它资源

- [什么是 kernel-core、kernel-modules 和 kernel-modules-extras 软件包？](#)

1.4. 显示内核软件包的内容

要确定内核软件包是否提供了特定文件，如模块，您可以通过查询存储库来显示用于架构的软件包的文件列表。不必下载或安装软件包来显示文件列表。

使用 **dnf** 工具查询文件列表，例如 **kernel-core**、**kernel-modules-core** 或 **kernel-modules** 软件包的文件列表。请注意，**kernel** 软件包是一个不包含任何文件的元软件包。

流程

1. 列出软件包的可用版本：

```
$ yum repoquery <package_name>
```

例如，列出 **kernel-core** 软件包的可用版本：

```
$ yum repoquery kernel-core
kernel-core-0:4.18.0-147.0.2.el8_1.x86_64
kernel-core-0:4.18.0-147.0.3.el8_1.x86_64
kernel-core-0:4.18.0-147.3.1.el8_1.x86_64
```

```
kernel-core-0:4.18.0-147.5.1.el8_1.x86_64
kernel-core-0:4.18.0-147.8.1.el8_1.x86_64
kernel-core-0:4.18.0-147.el8.x86_64
...
```

2. 显示软件包中文件的列表：

```
$ yum repoquery -l <package_name>
```

例如，显示 `kernel-core-0:5.14.0-162.23.1.el9_1.x86_64` 软件包中文件的列表。

```
$ yum repoquery -l kernel-core-0:4.18.0-147.8.1.el8_1.x86_64
/boot/.vmlinuz-4.18.0-147.8.1.el8_1.x86_64.hmac
/boot/System.map-4.18.0-147.8.1.el8_1.x86_64
/boot/config-4.18.0-147.8.1.el8_1.x86_64
/boot/initramfs-4.18.0-147.8.1.el8_1.x86_64.img
/boot/symvers-4.18.0-147.8.1.el8_1.x86_64.gz
/boot/vmlinuz-4.18.0-147.8.1.el8_1.x86_64
/etc/ld.so.conf.d/kernel-4.18.0-147.8.1.el8_1.x86_64.conf
/lib/modules
/lib/modules/4.18.0-147.8.1.el8_1.x86_64
...
```

其它资源

- [打包和分发软件](#)

1.5. 安装特定的内核版本

使用 `yum` 软件包管理器安装新内核。

流程

- 要安装特定的内核版本，请输入以下命令：

```
# yum install kernel-{version}
```

其它资源

- [Red Hat Code Browser](#)
- [Red Hat Enterprise Linux 发行日期](#)

1.6. 更新内核

使用 `yum` 软件包管理器更新内核。

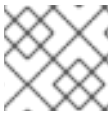
流程

1. 要更新内核，请输入以下命令：

```
# yum update kernel
```

此命令将内核以及所有依赖项更新至最新可用版本。

2. 重启您的系统以使更改生效。



注意

当从 RHEL 7 升级到 RHEL 8 时，请遵循 [从 RHEL 7 升级到 RHEL 8](#) 文档中的相关部分。

其它资源

- [管理软件包](#)

1.7. 将内核设置为默认

使用 **grubby** 命令行工具和 GRUB 将特定内核设置为默认。

流程

- 使用 **grubby** 工具将内核设置为默认
 - 输入以下命令，使用 **grubby** 工具将内核设置为默认：

```
# grubby --set-default $kernel_path
```

命令使用不带 **.conf** 后缀的计算机 ID 作为参数。



注意

机器 ID 位于 **/boot/loader/entries/** 目录中。

- 使用 **id** 参数将内核设置为默认
 - 使用 **id** 参数列出引导条目，然后将所需的内核设置为默认：

```
# grubby --info ALL | grep id
# grubby --set-default /boot/vmlinuz-<version>.<architecture>
```



注意

要使用 **title** 参数列出引导条目，请执行 **# grubby --info=ALL | grep title** 命令。

- 仅为下次引导设定默认内核
 - 执行以下命令，仅在下次使用 **grub2-reboot** 命令重新引导时设置默认内核：

```
# grub2-reboot <index|title|id>
```



警告

小心地为下次启动设置默认内核。安装新内核 RPM 的自构建内核并手动将这些条目添加到 `/boot/loader/entries/` 目录可能会更改索引值。

第 2 章 管理内核模块

了解内核模块、如何显示其信息，以及如何使用内核模块执行基本的管理任务。

2.1. 内核模块简介

Red Hat Enterprise Linux 内核可使用一个可选的、带有额外功能的模块（称为内核模块）进行扩展，而无需重启系统。在 RHEL 8 中，内核模块是构建到压缩的 `<KERNEL_MODULE_NAME>.ko.xz` 对象文件中的额外的内核代码。

内核模块启用的最常见功能是：

- 添加用于支持新硬件的设备驱动程序
- 支持文件系统，如 GFS2 或者 NFS
- 系统调用

在现代系统中，在需要时会自动载入内核模块。但在某些情况下，需要手动加载或卸载模块。

与内核本身一样，模块也可以在需要时采用自定义其行为的参数。

同时，提供了用来检查当前运行了哪些模块、哪些模块可以加载到内核以及模块接受哪些参数的工具。该工具还提供了在运行的内核中载入和卸载内核模块的机制。

2.2. 内核模块依赖关系

某些内核模块有时依赖一个或多个内核模块。`/lib/modules/<KERNEL_VERSION>/modules.dep` 文件包含对应内核版本的完整内核模块依赖关系列表。

depmod

依赖项文件由 `depmod` 程序生成，该程序是 `kmod` 软件包的一部分。`kmod` 提供的许多工具在执行操作时会考虑模块依赖关系，因此很少需要手动跟踪依赖项。



警告

内核模块的代码在内核空间中是在不受限制模式下执行的。因此，您应该了解您载入的模块。

weak-modules

除了 `depmod` 外，Red Hat Enterprise Linux 还提供了 `weak-modules` 脚本，也带有 `kmod` 软件包。`weak-modules` 决定哪些模块与安装的内核兼容 kABI。在检查模块内核的兼容性时，`weak-modules` 按照它们构建的内核的从高到低版本处理符号依赖项。这意味着 `weak-modules` 不依赖于它们构建的内核版本来处理每个模块。

其它资源

- `modules.dep (5)` 手册页

- [depmod \(8\) 手册页](#)
- [与 Red Hat Enterprise Linux 提供的 weak-modules 脚本的作用是什么？](#)
- [什么是内核应用程序二进制接口\(kABI\)？](#)

2.3. 列出已安装的内核模块

`grubby --info=ALL` 命令显示在 **!BLS** 和 **BLS** 安装中安装的内核的一个索引列表。

流程

- 使用以下命令列出安装的内核：

```
# grubby --info=ALL | grep title
```

下面是所有安装的内核列表：

```
title=Red Hat Enterprise Linux (4.18.0-20.el8.x86_64) 8.0 (Ootpa)
title=Red Hat Enterprise Linux (4.18.0-19.el8.x86_64) 8.0 (Ootpa)
title=Red Hat Enterprise Linux (4.18.0-12.el8.x86_64) 8.0 (Ootpa)
title=Red Hat Enterprise Linux (4.18.0) 8.0 (Ootpa)
title=Red Hat Enterprise Linux (0-rescue-2fb13ddde2e24fde9e6a246a942caed1) 8.0 (Ootpa)
```

以上示例显示了 GRUB 菜单中安装的 `grubby-8.40-17` 的内核列表。

2.4. 列出当前载入的内核模块

查看当前载入的内核模块。

先决条件

- 已安装 **kmod** 软件包。

流程

- 要列出所有当前载入的内核模块，请输入：

```
$ lsmod

Module              Size Used by
fuse                 126976 3
uinput               20480 1
xt_CHECKSUM          16384 1
ipt_MASQUERADE       16384 1
xt_contrack          16384 1
ipt_REJECT            16384 1
nft_counter          16384 16
nf_nat_tftp           16384 0
nf_contrack_tftp     16384 1 nf_nat_tftp
tun                   49152 1
bridge               192512 0
stp                   16384 1 bridge
```

```
llc          16384 2 bridge,stp
nf_tables_set 32768 5
nft_fib_inet 16384 1
...
```

在上例中：

- Module** 列提供了当前载入的模块的 **名称**。
- Size** 列显示每个模块的 **内存量**（以 KB 为单位）。
- Used by** 列显示 **依赖于** 特定模块的模块的编号，以及名称（可选）。

其它资源

- `/usr/share/doc/kmod/README` 文件
- `lsmod (8)` 手册页

2.5. 列出所有安装的内核

使用 `grubby` 实用程序列出系统上所有安装的内核。

先决条件

- 您有 `root` 权限。

流程

- 要列出所有安装的内核，请输入：

```
# grubby --info=ALL | grep ^kernel

kernel="/boot/vmlinuz-4.18.0-305.10.2.el8_4.x86_64"
kernel="/boot/vmlinuz-4.18.0-240.el8.x86_64"
kernel="/boot/vmlinuz-0-rescue-41eb2e172d7244698abda79a51778f1b"
```

输出显示了所有安装的内核的路径，也显示其各自的版本。

2.6. 显示内核模块信息

使用 `modinfo` 命令显示指定内核模块的一些详细信息。

先决条件

- 已安装 `kmod` 软件包。

流程

- 要显示关于任何内核模块的信息，请输入：

```
$ modinfo <KERNEL_MODULE_NAME>
```

例如：

```
$ modinfo virtio_net

filename:    /lib/modules/4.18.0-94.el8.x86_64/kernel/drivers/net/virtio_net.ko.xz
license:     GPL
description: Virtio network driver
rhelversion: 8.1
srcversion:  2E9345B281A898A91319773
alias:       virtio:d00000001v*
depends:      net_failover
intree:      Y
name:        virtio_net
vermagic:    4.18.0-94.el8.x86_64 SMP mod_unload modversions
...
parm:        napi_weight:int
parm:        csum:bool
parm:        gso:bool
parm:        napi_tx:bool
```

您可以查询所有可用模块的信息，无论它们是否被加载。**parm** 条目显示用户可以为模块设置的参数，以及它们预期的值类型。



注意

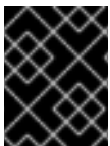
在输入内核模块的名称时，不要将 **.ko.xz** 扩展附加到名称的末尾。内核模块名称没有扩展名，它们对应的文件有。

其它资源

- **modinfo (8)** 手册页

2.7. 在系统运行时载入内核模块

扩展 Linux 内核功能的最佳方法是加载内核模块。使用 **modprobe** 命令查找并将内核模块载入到当前运行的内核中。



重要

重启系统后，这个过程中描述的更改**不会保留**。有关如何在系统重启后将内核模块载入为持久性的详情，请参考 [在系统引导时自动载入内核模块](#)。

先决条件

- 根权限
- 已安装 **kmod** 软件包。
- 相关的内核模块没有被加载。要确定是这种情况，请列出 [载入的内核模块](#)。

流程

1. 选择您要载入的内核模块。
模块位于 `/lib/modules/$(uname -r)/kernel/<SUBSYSTEM>/` 目录中。

2. 载入相关内核模块：

```
# modprobe <MODULE_NAME>
```



注意

在输入内核模块的名称时，不要将 **.ko.xz** 扩展附加到名称的末尾。内核模块名称没有扩展名，它们对应的文件有。

验证

- （可选）验证载入了相关模块：

```
$ lsmod | grep <MODULE_NAME>
```

如果正确加载了模块，这个命令会显示相关的内核模块。例如：

```
$ lsmod | grep serio_raw
serio_raw          16384 0
```

其它资源

- [modprobe \(8\) 手册页](#)

2.8. 在系统运行时卸载内核模块

有时，您发现您需要从运行的内核中卸载某些内核模块。使用 **modprobe** 命令在系统运行时查找并从当前加载的内核中卸载内核模块。



警告

当它们被正在运行的系统使用时，不要卸载这些内核模块。这样做可能会导致不稳定，或系统无法正常操作。



重要

完成这个过程后，被定义为在引导是自动加载的内核模块，在重启系统后将不会处于卸载状态。有关如何解决这个结果的详情，请参考 [防止在系统引导时自动载入内核模块](#)。

先决条件

- 根权限
- 已安装 **kmod** 软件包。

流程

1. 列出所有载入的内核模块：

```
# lsmod
```

2. 选择您要卸载的内核模块。
如果内核模块有依赖项，请在卸载内核模块前卸载它们。有关识别带有依赖项的模块的详情，请参阅 [列出当前载入的内核模块](#) 和 [内核模块依赖项](#)。
3. 卸载相关内核模块：

```
# modprobe -r <MODULE_NAME>
```

在输入内核模块的名称时，不要将 `.ko.xz` 扩展附加到名称的末尾。内核模块名称没有扩展名，它们对应的文件有。

验证

- （可选）验证相关模块是否已卸载：

```
$ lsmod | grep <MODULE_NAME>
```

如果模块被成功卸载，这个命令不会显示任何输出。

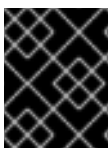
其它资源

- [modprobe\(8\) 手册页](#)

2.9. 在启动过程早期卸载内核模块

在某些情况下，需要在引导过程的早期卸载内核模块。例如，当内核模块包含一个代码时，会导致系统变得无响应，用户无法到达该阶段来永久禁用恶意内核模块。在这种情况下，使用引导装载程序可以临时阻止加载内核模块。

您可以编辑相关的引导装载程序条目，来在引导序列继续之前卸载所需的内核模块。



重要

此流程中描述的更改在下次重启后 **不会持久存在**。有关如何将内核模块添加到 `denylist` 中，以便在引导过程中不会被自动载入，请参阅 [防止在系统引导时自动载入内核模块](#)。

先决条件

- 您有一个可加载的内核模块，但出于某种原因您要防止其加载。

流程

1. 将系统启动到引导装载程序中。
2. 使用光标键突出显示相关的引导装载程序条目。
3. 按 **e** 键编辑条目。

图 2.1. 内核引导菜单

```

Red Hat Enterprise Linux (4.18.0-305.10.2.el8_4.x86_64) 8.4 (Ootpa)
Red Hat Enterprise Linux (4.18.0-305.3.1.el8_4.x86_64) 8.4 (Ootpa)
Red Hat Enterprise Linux (4.18.0-240.22.1.el8_3.x86_64) 8.3 (Ootpa)
Red Hat Enterprise Linux (0-rescue-bfeb014063334b2da66b978e95804445) 8.3->

Use the ↑ and ↓ keys to change the selection.
Press 'e' to edit the selected item, or 'c' for a command prompt.

```

4. 使用光标键导航到以 `linux` 开头的行。
5. 将 `modprobe.blacklist=module_name` 附加到行尾。

图 2.2. 内核引导条目

```

load_video
set gfx_payload=keep
insmod gzio
linux ($root)/vmlinuz-4.18.0-305.10.2.el8_4.x86_64 root=/dev/mapper/rhel-root \
ro crashkernel=auto resume=/dev/mapper/rhel-swap rd.lvm.lv=rhel/root rd.lvm.lv\
=rhel/swap rhgb quiet modprobe.blacklist=serio_raw
initrd ($root)/initramfs-4.18.0-305.10.2.el8_4.x86_64.img $tuned_initrd

Press Ctrl-x to start, Ctrl-c for a command prompt or Escape to
discard edits and return to the menu. Pressing Tab lists
possible completions.

```

`serio_raw` 内核模块演示了在引导过程早期卸载的恶意模块。

6. 按 **Ctrl+X** 使用修改后的配置启动。

验证

- 系统完全启动后，验证相关的内核模块是否没有加载。

```
# lsmod | grep serio_raw
```

其它资源

- [管理内核模块](#)

2.10. 在系统引导时自动载入内核模块

配置一个内核模块以便在引导过程中自动载入该模块。

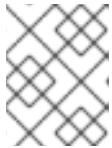
先决条件

- 根权限
- 已安装 **kmod** 软件包。

流程

1. 选择您要在引导过程中载入的内核模块。
模块位于 `/lib/modules/$(uname -r)/kernel/<SUBSYSTEM>/` 目录中。
2. 为模块创建配置文件：

```
# echo <MODULE_NAME> > /etc/modules-load.d/<MODULE_NAME>.conf
```



注意

在输入内核模块的名称时，不要将 **.ko.xz** 扩展附加到名称的末尾。内核模块名称没有扩展名，它们对应的文件有。

3. 另外，重启后，验证载入了相关模块：

```
$ lsmod | grep <MODULE_NAME>
```

上面的示例命令应该成功并显示相关的内核模块。



重要

重启系统后，这个过程中描述的更改将会保留。

其它资源

- [modules-load.d\(5\) 手册页](#)

2.11. 防止在系统引导时自动载入内核模块

您可以通过使用相应的命令在 **modprobe** 配置文件中列出模块，来防止系统在引导过程中自动载入内核模块。

先决条件

- 此流程中的命令需要 **root** 权限。使用 **su -** 切换到 **root** 用户，或使用 **sudo** 作为命令的前导。
- 已安装 **kmod** 软件包。
- 确定您当前的系统配置不需要您计划拒绝的内核模块。

流程

1. 使用 **lsmod** 命令列出载入到当前运行的内核中的模块：


```
$ lsmod
Module          Size Used by
tls             131072 0
uinput         20480 1
snd_seq_dummy  16384 0
snd_hrtimer    16384 1
...
```

在输出中，识别您要防止被加载的模块。

- 或者，识别您要防止在 `/lib/modules/<KERNEL-VERSION>/kernel/<SUBSYSTEM>/` 目录中加载的而未加载的内核模块，例如：

```
$ ls /lib/modules/4.18.0-477.20.1.el8_8.x86_64/kernel/crypto/
ansi_cprng.ko.xz  chacha20poly1305.ko.xz  md4.ko.xz
serpent_generic.ko.xz
anubis.ko.xz      cmac.ko.xz...
```

2. 创建一个作为 denylist 的配置文件：

```
# touch /etc/modprobe.d/denylist.conf
```

3. 在您选择的文本编辑器中，将您要从自动加载中排除的模块名称与使用 **blacklist** 配置命令列出的模块名称结合在一起，例如：

```
# Prevents <KERNEL-MODULE-1> from being loaded
blacklist <MODULE-NAME-1>
install <MODULE-NAME-1> /bin/false

# Prevents <KERNEL-MODULE-2> from being loaded
blacklist <MODULE-NAME-2>
install <MODULE-NAME-2> /bin/false
...
```

由于 **blacklist** 命令不会阻止将模块作为不在 denylist 中的另一个内核模块的依赖项加载，所以您还必须定义 **install** 行。在这种情况下，系统运行 `/bin/false`，而不是安装模块。以哈希符号开头的行是注释，可用于使文件更易读。



注意

在输入内核模块的名称时，不要将 **.ko.xz** 扩展附加到名称的末尾。内核模块名称没有扩展名，它们对应的文件有。

4. 在重建前，创建当前初始 RAM 磁盘镜像的一个备份副本：

```
# cp /boot/initramfs-$(uname -r).img /boot/initramfs-$(uname -r).bak.$(date +%m-%d-%H%M%S).img
```

- 或者，创建初始 RAM 磁盘镜像的一个备份副本，该镜像对应于您要阻止内核模块自动载入的内核版本：

```
# cp /boot/initramfs-<VERSION>.img /boot/initramfs-<VERSION>.img.bak.$(date +%m-%d-%H%M%S)
```

5. 生成一个新的初始 RAM 磁盘镜像以应用更改：

```
# dracut -f -v
```

- 如果您为系统当前使用的内核版本之外的版本构建初始 RAM 磁盘镜像，请指定目标 **initramfs** 和内核版本：

```
# dracut -f -v /boot/initramfs-<TARGET-VERSION>.img <CORRESPONDING-TARGET-KERNEL-VERSION>
```

6. 重启系统：

```
$ reboot
```



重要

此流程中描述的更改将在重启后**生效并保留**。如果您在 `denylist` 中错误地列出了关键内核模块，您可以将系统切换到不稳定或不正常工作的状态。

其它资源

- [如何防止内核模块自动加载？](#) 解决方案文章
- **modprobe.d (5)** 和 **dracut (8)** 手册页

2.12. 编译自定义的内核模块

您可以根据硬件和软件级别的各种配置的要求构建一个采样内核模块。

先决条件

- **kernel-devel**、**gcc** 和 **elfutils-libelf-devel** 软件包已安装。

```
# dnf install kernel-devel-$(uname -r) gcc elfutils-libelf-devel
```

- 您有 root 权限。
- 您创建了 **/root/testmodule/** 目录,在此编译自定义的内核模块。

流程

1. 创建包含以下内容的 **/root/testmodule/test.c** 文件：

```
#include <linux/module.h>
#include <linux/kernel.h>

int init_module(void)
{ printk("Hello World\n This is a test\n"); return 0; }

void cleanup_module(void)
{ printk("Good Bye World"); }
```

test.c 文件是一个源文件，其向内核模块提供主要功能。出于组织需要，该文件已创建在专用的 **/root/testmodule/** 目录中。在模块编译后，**/root/testmodule/** 目录将包含多个文件。

test.c 文件包含来自系统库的文件：

- 示例代码中的 **printk ()** 函数需要 **linux/kernel.h** 头文件。
 - **linux/module.h** 文件包含可在用 C 编程语言编写的多个源文件之间共享的函数声明和宏定义。
2. 按照 **init_module ()** 和 **cleanup_module ()** 函数启动和结束内核日志记录函数 **printk ()**，后者会打印文本。
 3. 使用以下内容创建 **/root/testmodule/Makefile** 文件。

```
obj-m := test.o
```

Makefile 包含编译器必须生成一个具体名为 **test.o** 的对象文件的指令。**obj-m** 指令指定生成的 **test.ko** 文件将编译为可加载的内核模块。或者，**obj-y** 指令会指示将 **test.ko** 构建为一个内置内核模块。

4. 编译内核模块。

```
# make -C /lib/modules/$(uname -r)/build M=/root/testmodule modules
make: Entering directory '/usr/src/kernels/4.18.0-305.el8.x86_64'
  CC [M] /root/testmodule/test.o
  Building modules, stage 2.
  MODPOST 1 modules
WARNING: modpost: missing MODULE_LICENSE() in /root/testmodule/test.o
see include/linux/module.h for more information
  CC /root/testmodule/test.mod.o
  LD [M] /root/testmodule/test.ko
make: Leaving directory '/usr/src/kernels/4.18.0-305.el8.x86_64'
```

编译器将它们链接成最终内核模块(**test.ko**)之前，会为每个源文件(**test.c**)创建一个对象文件(**test.o**)来作为中间步骤。

成功编译后，**/root/testmodule/** 包含与编译的自定义内核模块相关的其他文件。已编译的模块本身由 **test.ko** 文件表示。

验证

1. 可选：检查 **/root/testmodule/** 目录的内容：

```
# ls -l /root/testmodule/
total 152
-rw-r--r--. 1 root root 16 Jul 26 08:19 Makefile
-rw-r--r--. 1 root root 25 Jul 26 08:20 modules.order
-rw-r--r--. 1 root root 0 Jul 26 08:20 Module.symvers
-rw-r--r--. 1 root root 224 Jul 26 08:18 test.c
-rw-r--r--. 1 root root 62176 Jul 26 08:20 test.ko
-rw-r--r--. 1 root root 25 Jul 26 08:20 test.mod
-rw-r--r--. 1 root root 849 Jul 26 08:20 test.mod.c
-rw-r--r--. 1 root root 50936 Jul 26 08:20 test.mod.o
-rw-r--r--. 1 root root 12912 Jul 26 08:20 test.o
```

2. 将内核模块复制到 `/lib/modules/$(uname -r)/` 目录中：

```
# cp /root/testmodule/test.ko /lib/modules/$(uname -r)/
```

3. 更新模块依赖项列表：

```
# depmod -a
```

4. 载入内核模块：

```
# modprobe -v test  
insmod /lib/modules/4.18.0-305.el8.x86_64/test.ko
```

5. 验证内核模块是否成功载入：

```
# lsmod | grep test  
test                16384 0
```

6. 从内核环缓冲中读取最新的消息：

```
# dmesg  
[74422.545004] Hello World  
                This is a test
```

其它资源

- [管理内核模块](#)

第 3 章 为安全引导签名内核和模块

您可以使用签名的内核和签名的内核模块来加强系统的安全性。在启用了安全引导机制的基于 UEFI 的构建系统中，您可以自我签名一个私有构建的内核或内核模块。另外，您可以将公钥导入到要部署内核或内核模块的目标系统中。

如果启用了安全引导机制，则必须使用私钥签名以下所有组件，并使用对应的公钥进行身份验证：

- UEFI 操作系统引导装载程序
- Red Hat Enterprise Linux 内核
- 所有内核模块

如果这些组件中的任何一个都没有签名和验证，则系统将无法完成引导过程。

RHEL 8 包括：

- 签名的引导装载程序
- 签名的内核
- 签名的内核模块

此外，签名的第一阶段引导装载程序和签名的内核包括嵌入的红帽公钥。这些签名的可执行二进制文件和嵌入式密钥可让 RHEL 8 在支持 UEFI 安全引导的系统上使用 UEFI 固件提供的 Microsoft UEFI 安全引导认证认证机构密钥安装、启动并运行。



注意

- 不是所有基于 UEFI 的系统都包括对安全引导的支持。
- 构建系统（构建和签署内核模块）不需要启用 UEFI 安全引导，甚至不需要是基于 UEFI 的系统。

3.1. 先决条件

- 要能够为外部构建的内核模块签名，请从以下软件包安装工具：

```
# yum install pesign openssl kernel-devel mokutil keyutils
```

表 3.1. 所需工具

工具	由软件包提供	用于	目的
efikeygen	pesign	构建系统	生成公共和专用 X.509 密钥对
openssl	openssl	构建系统	导出未加密的私钥
sign-file	kernel-devel	构建系统	用来使用私钥为内核模块签名的可执行文件
mokutil	mokutil	目标系统	用于手动注册公钥的可选工具

工具	由软件包提供	用于	目的
keyctl	keyutils	目标系统	用于在系统密钥环中显示公钥的可选工具

3.2. 什么是 UEFI 安全引导

使用 *统一可扩展固件接口* (UEFI) 安全引导技术，您可以防止未由可信密钥签名的内核空间代码的执行。系统引导装载程序使用加密密钥进行签名。公钥的数据库（其包含在固件中）授权签名密钥。然后，您可以在下一个阶段引导装载程序和内核中验证签名。

UEFI 安全引导建立了一个从固件到签名驱动程序和内核模块的信任链，如下所示：

- UEFI 私钥签名，公钥验证 **shim** 第一阶段引导装载程序。*证书颁发机构* (CA) 反过来签署公钥。CA 存储在固件数据库中。
- **shim** 文件包含红帽公钥 **Red Hat Secure Boot (CA 密钥 1)** 来验证 GRUB 引导装载程序和内核。
- 内核又包含用于验证驱动程序和模块的公钥。

安全引导是 UEFI 规范的引导路径验证组件。规范定义：

- 用于非易失性存储中加密保护的 UEFI 变量的编程接口。
- 在 UEFI 变量中存储可信的 X.509 根证书。
- UEFI 应用程序的验证，如引导装载程序和驱动程序。
- 撤销已知错误的证书和应用程序哈希的流程。

UEFI 安全引导版主可以检测未经授权的更改，但不会：

- 防止安装或删除第二阶段引导装载程序。
- 需要用户明确确认此类更改。
- 停止引导路径操作。在引导过程中会验证签名，而不是在安装或更新引导装载程序时。

如果引导装载程序或内核不是由系统可信密钥签名的，则安全引导会阻止它们启动。

3.3. UEFI 安全引导支持

如果内核和所有载入的驱动程序都使用可信密钥签名了，您可以在启用了 UEFI 安全引导的系统上安装并运行 RHEL 8。红帽提供了由相关红帽密钥签名和验证的内核和驱动程序。

如果要加载外部构建的内核或驱动程序，还必须给它们签名。

UEFI 安全引导施加的限制

- 系统仅在签名被正确验证后才运行 kernel-mode 代码。
- GRUB 模块加载被禁用，因为没有签名和验证 GRUB 模块的基础架构。以便加载它们构成了在安全引导定义的安全范围内不受信任的代码的执行。

- 红帽提供了一个签名的 GRUB 二进制文件，其中包含 RHEL 8 上所有支持的模块。

其它资源

- [UEFI 安全引导施加的限制](#)

3.4. 使用 X.509 密钥验证内核模块的要求

在 RHEL 8 中，当加载内核模块时，内核会根据内核系统密钥环中的公共 X.509 密钥检查模块的签名 (`.builtin_trusted_keys`) 以及内核平台密钥环 (`.platform`)。 `.platform` 密钥环包含来自第三方平台提供商和自定义公钥的密钥。内核系统 `.blacklist` 密钥环中的密钥不包括在验证中。

您需要满足某些条件，才能在启用了 UEFI 安全引导功能的系统上载入内核模块：

- 如果启用了 UEFI 安全引导，或者指定了 `module.sig_enforce` 内核参数：
 - 您只能加载那些签名是通过系统密钥环 (`.builtin_trusted_keys`) 和平台密钥环 (`.platform`) 验证的已签名内核模块。
 - 公钥不能在系统中被撤销的密钥环 (`.blacklist`)。
- 如果禁用了 UEFI 安全引导且未指定 `module.sig_enforce` 内核参数：
 - 您可以加载未签名的内核模块和签名的内核模块，而无需公钥。
- 如果系统不基于 UEFI，或者禁用 UEFI 安全引导：
 - 只有内核中嵌入的密钥才会加载到 `.builtin_trusted_keys` 和 `.platform`。
 - 您无法在不重新构建内核的情况下添加这组密钥。

表 3.2. 加载内核模块的验证要求

模块已签名	找到公钥，且签名有效	UEFI 安全引导状态	sig_enforce	模块载入	内核污点
未签名	-	未启用	未启用	成功	是
		未启用	Enabled	Fails	-
		Enabled	-	Fails	-
已签名	否	未启用	未启用	成功	是
		未启用	Enabled	Fails	-
		Enabled	-	Fails	-
已签名	是	未启用	未启用	成功	否
		未启用	Enabled	成功	否
		未启用	Enabled	成功	否

模块已签名	找到公钥，且签名有效	UEFI 安全引导状态	sig_enforce	模块载入	内核污点
		Enabled	-	成功	否

3.5. 公钥的源

在引导过程中，内核会从一组持久性密钥中加载 X.509 密钥到以下密钥环中：

- 系统密钥环 (**.builtin_trusted_keys**)
- **.platform** 密钥环
- 系统 **.blacklist** 密钥环

表 3.3. 系统密钥环源

X.509 密钥源	用户可以添加密钥	UEFI 安全引导状态	引导过程中载入的密钥
嵌入于内核中	否	-	.builtin_trusted_keys
UEFI db	有限	未启用	否
		Enabled	.platform
嵌入在 shim 引导装载程序中	否	未启用	否
		Enabled	.platform
Machine Owner Key (MOK) 列表	是	未启用	否
		Enabled	.platform

.builtin_trusted_keys

- 在引导时构建的密钥环
- 包含可信公钥
- 查看密钥需要 **root** 权限

.platform

- 在引导时构建的密钥环
- 包含来自第三方平台提供商和自定义公钥的密钥
- 查看密钥需要 **root** 权限

.blacklist

- 具有已撤销的 X.509 密钥的密钥环
- 使用来自 **.blacklist** 的密钥签名的模块将会验证失败，即使您的公钥在 **.builtin_trusted_keys** 中

UEFI 安全引导 db

- 签名数据库
- 存储 UEFI 应用程序、UEFI 驱动程序和引导装载程序的密钥（哈希）
- 密钥可在机器上加载

UEFI 安全引导 dbx

- 已撤销的签名数据库
- 防止密钥被加载
- 来自此数据库的撤销的密钥被添加到 **.blacklist** 密钥环中

3.6. 生成公钥和私钥对

要在启用了安全引导的系统上使用自定义内核或自定义内核模块，您必须生成一个公钥和私有 X.509 密钥对。您可以使用生成的私钥为内核或内核模块签名。您还可以通过在向安全引导的 Machine Owner Key (MOK) 中添加相应的公钥来验证签名的内核或内核模块。



警告

应用强大的安全措施和访问策略来保护您的私钥内容。对于一个恶意的用户，可以使用这个密钥破坏所有由对应公钥验证的系统。

流程

- 创建一个 X.509 公钥和私钥对：
 - 如果您只想为自定义内核 *模块* 签名：

```
# efkeygen --dbdir /etc/pki/pesign \
  --self-sign \
  --module \
  --common-name 'CN=Organization signing key' \
  --nickname 'Custom Secure Boot key'
```

- 如果要为自定义 *内核* 签名：

```
# efkeygen --dbdir /etc/pki/pesign \
  --self-sign \
  --kernel \
```

```
--common-name 'CN=Organization signing key' \  
--nickname 'Custom Secure Boot key'
```

- 当 RHEL 系统运行 FIPS 模式时：

```
# efkeygen --dbdir /etc/pki/pesign \  
--self-sign \  
--kernel \  
--common-name 'CN=Organization signing key' \  
--nickname 'Custom Secure Boot key' \  
--token 'NSS FIPS 140-2 Certificate DB'
```



注意

在 FIPS 模式下，您必须使用 **--token** 选项，以便 **efkeygen** 在 PKI 数据库中找到默认的“NSS Certificate DB”令牌。

公钥和私钥现在存储在 `/etc/pki/pesign/` 目录中。



重要

好的安全实践是在签名密钥的有效周期内为内核和内核模块签名。但是，**sign-file** 工具不会警告您，无论有效期日期是什么，密钥在 RHEL 8 中都可用。

其它资源

- [openssl\(1\) 手册页](#)
- [RHEL 安全指南](#)
- [通过在 MOK 列表中添加公钥在目标系统中注册公钥](#)

3.7. 系统密钥环输出示例

您可以使用 **keyutils** 软件包中的 **keyctl** 工具显示系统密钥环上的密钥信息。

先决条件

- 您有 root 权限。
- 您已从 **keyutils** 软件包中安装了 **keyctl** 工具。

例 3.1. 密钥环输出

以下是启用了 UEFI 安全引导的 RHEL 8 系统中的 **.builtin_trusted_keys**、**.platform** 和 **.blacklist** 密钥环的简化示例输出。

```
# keyctl list %:.builtin_trusted_keys  
6 keys in keyring:  
...asymmetric: Red Hat Enterprise Linux Driver Update Program (key 3): bf57f3e87...  
...asymmetric: Red Hat Secure Boot (CA key 1): 4016841644ce3a810408050766e8f8a29...  
...asymmetric: Microsoft Corporation UEFI CA 2011: 13adb4309bd82709c8cd54f316ed...  
...asymmetric: Microsoft Windows Production PCA 2011: a92902398e16c49778cd90f99e...
```

```
...asymmetric: Red Hat Enterprise Linux kernel signing key: 4249689eefc77e95880b...
...asymmetric: Red Hat Enterprise Linux kpatch signing key: 4d38fd864ebe18c5f0b7...
```

```
# keyctl list %:.platform
```

```
4 keys in keyring:
```

```
...asymmetric: VMware, Inc.: 4ad8da0472073...
...asymmetric: Red Hat Secure Boot CA 5: cc6fafa72...
...asymmetric: Microsoft Windows Production PCA 2011: a929f298e1...
...asymmetric: Microsoft Corporation UEFI CA 2011: 13adbf4e0bd82...
```

```
# keyctl list %:.blacklist
```

```
4 keys in keyring:
```

```
...blacklist: bin:f5ff83a...
...blacklist: bin:0dfdbec...
...blacklist: bin:38f1d22...
...blacklist: bin:51f831f...
```

示例中的 `.builtin_trusted_keys` 密钥环显示从 UEFI 安全引导 `db` 密钥以及 **Red Hat Secure Boot (CA 密钥 1)** (其嵌入在 `shim` 引导装载程序中) 中添加两个密钥。

例 3.2. 内核控制台输出

以下示例显示了内核控制台的输出结果。消息标识带有 UEFI 安全引导相关源的密钥。这包括 UEFI 安全引导 `db`、嵌入的 `shim` 和 MOK 列表。

```
# dmesg | egrep 'integrity.*cert'
```

```
[1.512966] integrity: Loading X.509 certificate: UEFI:db
[1.513027] integrity: Loaded X.509 cert 'Microsoft Windows Production PCA 2011: a929023...
[1.513028] integrity: Loading X.509 certificate: UEFI:db
[1.513057] integrity: Loaded X.509 cert 'Microsoft Corporation UEFI CA 2011: 13adbf4309...
[1.513298] integrity: Loading X.509 certificate: UEFI:MokListRT (MOKvar table)
[1.513549] integrity: Loaded X.509 cert 'Red Hat Secure Boot CA 5: cc6fa5e72868ba494e93...
```

其它资源

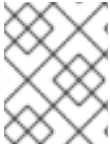
- [keyctl\(1\)、dmesg\(1\) 手册页](#)

3.8. 通过在 MOK 列表中添加公钥在目标系统中注册公钥

您必须在要验证并载入内核或内核模块的所有系统上注册您的公钥。您可以以不同的方式在目标系统上导入公钥，以便平台密钥环(`.platform`)能够使用公钥来验证内核或内核模块。

当 RHEL 8 在启用了安全引导机制的基于 UEFI 的系统上引导时，内核会将所有安全引导 `db` 密钥数据库中的公钥加载到平台密钥环(`.platform`)上。同时，内核排除了撤销的密钥的 `dbx` 数据库中的密钥。

您可以使用 Machine Owner Key (MOK)功能来扩展 UEFI 安全引导密钥数据库。当 RHEL 8 在启用了安全引导机制的 UEFI 系统上引导时，除了密钥数据库中的密钥外，MOK 列表中的密钥也会被添加到平台密钥环(`.platform`)中。和安全引导数据库密钥相似，MOK 列表密钥会被安全地永久存储。但它们是两个独立的工具。`shim`、`MokManager`、`GRUB` 和 `mokutil` 工具都支持 MOK 工具。



注意

为了便于对系统中的内核模块进行身份验证，请您的系统供应商将公钥合并到其工厂固件镜像中的 UEFI 安全引导密钥数据库中。

先决条件

- 您已生成了一个公钥和私钥对，并了解公钥的有效日期。详情请参阅[生成公钥和私钥对](#)。

流程

1. 将您的公钥导出到 **sb_cert.cer** 文件中：

```
# certutil -d /etc/pki/pesign \
  -n 'Custom Secure Boot key' \
  -Lr \
  > sb_cert.cer
```

2. 将您的公钥导入到 MOK 列表中：

```
# mokutil --import sb_cert.cer
```

3. 输入此 MOK 注册请求的新密码。

4. 重启机器。

shim 引导装载程序会注意到待处理的 MOK 密钥注册请求，并启动 **MokManager.efi**，以使您从 UEFI 控制台完成注册。

5. 选择 **Enroll MOK**，在提示时输入之前与此请求关联的密码，并确认注册。
您的公钥已添加到 MOK 列表中，这是永久的。

密钥位于 MOK 列表中后，它将会在启用 UEFI 安全引导时自动将其传播到此列表上的 **.platform** 密钥环中。

3.9. 使用私钥签名内核

如果启用了 UEFI 安全引导机制，您可以通过载入签名的内核在系统上获得增强的安全好处。

先决条件

- 您已生成了一个公钥和私钥对，并了解公钥的有效日期。详情请参阅[生成公钥和私钥对](#)。
- 您已在目标系统上注册了公钥。详情请查看[在 MOK 列表中添加公钥在目标系统中注册公钥](#)。
- 您有一个可用于签名的 ELF 格式的内核镜像。

流程

- 在 x64 构架上：

- a. 创建一个签名的镜像：

```
# pesign --certificate 'Custom Secure Boot key' \
  --in vmlinuz-version \
```

```
--sign \  
--out vmlinuz-version.signed
```

使用 **vmlinuz** 文件的版本后缀替换 **version**，使用您之前选择的名称替换 **Custom Secure Boot key**。

- b. 可选：检查签名：

```
# pesign --show-signature \  
--in vmlinuz-version.signed
```

- c. 使用签名镜像覆盖未签名的镜像：

```
# mv vmlinuz-version.signed vmlinuz-version
```

- 在 64 位 ARM 架构中：

- a. 解压缩 **vmlinuz** 文件：

```
# zcat vmlinuz-version > vmlinux-version
```

- b. 创建一个签名的镜像：

```
# pesign --certificate 'Custom Secure Boot key' \  
--in vmlinux-version \  
--sign \  
--out vmlinux-version.signed
```

- c. 可选：检查签名：

```
# pesign --show-signature \  
--in vmlinux-version.signed
```

- d. 压缩 **vmlinux** 文件：

```
# gzip --to-stdout vmlinux-version.signed > vmlinuz-version
```

- e. 删除未压缩的 **vmlinux** 文件：

```
# rm vmlinux-version*
```

3.10. 使用私钥签名 GRUB 构建

在启用了 UEFI 安全引导机制的系统上，您可以使用自定义的现有私钥为 GRUB 构建签名。如果您使用自定义 GRUB 构建，或者已经从系统中删除 Microsoft 信任锚，则您必须执行此操作。

先决条件

- 您已生成了一个公钥和私钥对，并了解公钥的有效日期。详情请参阅[生成公钥和私钥对](#)。
- 您已在目标系统上注册了公钥。详情请查看在[MOK 列表中添加公钥在目标系统中注册公钥](#)。
- 您有一个可用于签名的 GRUB EFI 二进制文件。

流程

- 在 x64 构架上：

- a. 创建一个签名的 GRUB EFI 二进制文件：

```
# pesign --in /boot/efi/EFI/redhat/grubx64.efi \  
--out /boot/efi/EFI/redhat/grubx64.efi.signed \  
--certificate 'Custom Secure Boot key' \  
--sign
```

将 **Custom Secure Boot key** 替换为您之前选择的名称。

- b. 可选：检查签名：

```
# pesign --in /boot/efi/EFI/redhat/grubx64.efi.signed \  
--show-signature
```

- c. 使用签名的二进制文件覆盖未签名的二进制文件：

```
# mv /boot/efi/EFI/redhat/grubx64.efi.signed \  
/boot/efi/EFI/redhat/grubx64.efi
```

- 在 64 位 ARM 架构中：

- a. 创建一个签名的 GRUB EFI 二进制文件：

```
# pesign --in /boot/efi/EFI/redhat/grubaa64.efi \  
--out /boot/efi/EFI/redhat/grubaa64.efi.signed \  
--certificate 'Custom Secure Boot key' \  
--sign
```

将 **Custom Secure Boot key** 替换为您之前选择的名称。

- b. 可选：检查签名：

```
# pesign --in /boot/efi/EFI/redhat/grubaa64.efi.signed \  
--show-signature
```

- c. 使用签名的二进制文件覆盖未签名的二进制文件：

```
# mv /boot/efi/EFI/redhat/grubaa64.efi.signed \  
/boot/efi/EFI/redhat/grubaa64.efi
```

3.11. 使用私钥签名内核模块

如果启用了 UEFI 安全引导机制，您可以通过加载签名的内核模块来提高系统的安全性。

在禁用了 UEFI 安全引导的系统上或非 UEFI 系统上，您签名的内核模块也是可以加载的。因此，您不需要提供内核模块的签名和未签名版本。

先决条件

- 您已生成了一个公钥和私钥对，并了解公钥的有效日期。详情请参阅[生成公钥和私钥对](#)。

- 您已在目标系统上注册了公钥。详情请查看在 [MOK 列表中添加公钥在目标系统中注册公钥](#)。
- 您有一个可以签注的 ELF 镜像格式的内核模块。

流程

1. 将您的公钥导出到 **sb_cert.cer** 文件中：

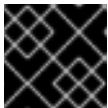
```
# certutil -d /etc/pki/pesign \
  -n 'Custom Secure Boot key' \
  -Lr \
  > sb_cert.cer
```

2. 从 NSS 数据库中提取密钥作为 PKCS #12 文件：

```
# pk12util -o sb_cert.p12 \
  -n 'Custom Secure Boot key' \
  -d /etc/pki/pesign
```

3. 当上一命令提示您时，输入加密了私钥的新密码。
4. 导出未加密的私钥：

```
# openssl pkcs12 \
  -in sb_cert.p12 \
  -out sb_cert.priv \
  -nocerts \
  -nodes
```



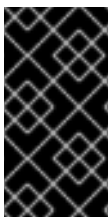
重要

请谨慎处理未加密的私钥。

5. 为内核模块签名。以下命令将签名直接附加到内核模块文件中的 ELF 镜像中：

```
# /usr/src/kernels/$(uname -r)/scripts/sign-file \
  sha256 \
  sb_cert.priv \
  sb_cert.cer \
  my_module.ko
```

您的内核模块现在可以被加载。



重要

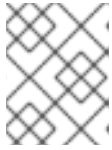
在 RHEL 8 中，密钥对的有效性日期非常重要。这个密钥没有过期，但必须在其签名密钥的有效周期内对内核模块进行签名。**sign-file** 实用程序不会提醒您这样做。例如：一个只在 2019 年有效的密钥可用来验证在 2019 年中使用该密钥签名的内核模块。但是，用户无法使用这个密钥在 2020 年签注一个内核模块。

验证

1. 显示关于内核模块签名的信息：

```
# modinfo my_module.ko | grep signer
signer:      Your Name Key
```

检查签名中是否列出了您在生成过程中输入的名称。



注意

附加的签名不包含在 ELF 镜像部分，不是 ELF 镜像的一个正式部分。因此，**readelf** 等工具无法在内核模块上显示签名。

2. 载入模块：

```
# insmod my_module.ko
```

3. 删除（未加载）模块：

```
# modprobe -r my_module.ko
```

其它资源

- [显示内核模块信息](#)

3.12. 载入经过签名的内核模块

一旦您的公钥注册到系统密钥环(**.builtin_trusted_keys**)和 MOK 列表中，在使用私钥签名的了相应的内核模块后，您可以使用 **modprobe** 命令加载签名的内核模块。

先决条件

- 您已生成了公钥和私钥对。详情请参阅[生成公钥和私钥对](#)。
- 您已经在系统密钥环中注册了公钥。详情请查看在[MOK 列表中添加公钥在目标系统中注册公钥](#)。
- 您用私钥签名了一个内核模块。详情请查看[使用私钥签名内核模块](#)。
- 安装 **kernel-modules-extra** 软件包，它会创建 **/lib/modules/\$(uname -r)/extra/** 目录：

```
# yum -y install kernel-modules-extra
```

流程

1. 验证您的公钥是否在系统密钥环中：

```
# keyctl list %:.platform
```

2. 将内核模块复制到您想要的内核的 **extra/** 目录中：

```
# cp my_module.ko /lib/modules/$(uname -r)/extra/
```

3. 更新模块依赖项列表：

■


```
# depmod -a
```

4. 载入内核模块：

```
# modprobe -v my_module
```

5. 另外，要在引导时载入模块，将其添加到 `/etc/modules-loaded.d/my_module.conf` 文件中：

```
# echo "my_module" > /etc/modules-load.d/my_module.conf
```

验证

- 验证模块是否被成功载入：

```
# lsmod | grep my_module
```

其它资源

- [管理内核模块](#)

第 4 章 配置内核命令行参数

使用内核命令行参数，您可以在引导时更改 Red Hat Enterprise Linux 内核某些方面的行为。作为系统管理员，您可以完全控制引导时要设置的选项。某些内核行为只能在引导时设置，因此了解如何进行这些更改是一项关键的管理技能。



重要

通过修改内核命令行参数更改系统的行为可能会对您的系统产生负面影响。始终在生产环境中部署更改前测试它们。如需进一步帮助，请联络红帽支持团队。

4.1. 什么是内核命令行参数

使用内核命令行参数，您可以覆盖默认值并设置特定的硬件设置。在引导时，您可以配置以下功能：

- Red Hat Enterprise Linux 内核
- 初始 RAM 磁盘
- 用户空间特性

默认情况下，使用 GRUB 引导装载程序的系统的内核命令行参数定义在每个内核引导条目的 `/boot/grub2/grubenv` 文件的 `kernelopts` 变量中。



注意

对于 IBM Z，内核命令行参数保存在引导条目配置文件中，因为 `zipl` 引导装载程序不支持环境变量。因此，无法使用 `kernelopts` 环境变量。

您可以使用 `grubby` 工具操作引导装载程序配置文件。使用 `grubby`，您可以执行以下操作：

- 更改默认的引导条目。
- 从 GRUB 菜单条目中添加或删除参数。

其它资源

- [kernel-command-line\(7\)、bootparam\(7\) 和 dracut.cmdline\(7\) 手册页](#)
- [如何在 Red Hat Enterprise Linux 8 中安装并引导自定义内核](#)
- [grubby \(8\) 手册页](#)

4.2. 了解引导条目

引导条目是保存在配置文件中并绑定到特定内核版本的选项集合。在实践中，您的引导条目至少与您所安装的系统数量相同。引导条目配置文件位于 `/boot/loader/entries/` 目录中，如下所示：

```
6f9cc9cb7d7845d49698c9537337cedc-4.18.0-5.el8.x86_64.conf
```

以上文件名由存储在 `/etc/machine-id` 文件中的计算机 ID 和内核版本组成。

引导条目配置文件包含有关内核版本、初始 ramdisk 镜像和 **kernelopts** 环境变量的信息，其中包含内核命令行参数。引导条目配置的示例内容如下：

```
title Red Hat Enterprise Linux (4.18.0-74.el8.x86_64) 8.0 (Ootpa)
version 4.18.0-74.el8.x86_64
linux /vmlinuz-4.18.0-74.el8.x86_64
initrd /initramfs-4.18.0-74.el8.x86_64.img $tuned_initrd
options $kernelopts $tuned_params
id rhel-20190227183418-4.18.0-74.el8.x86_64
grub_users $grub_users
grub_arg --unrestricted
grub_class kernel
```

kernelopts 环境变量在 `/boot/grub2/grubenv` 文件中定义。

其它资源

- [如何在 Red Hat Enterprise Linux 8 中安装并引导自定义内核](#)

4.3. 为所有引导条目更改内核命令行参数

更改系统上所有引导条目的内核命令行参数。

先决条件

- 验证 **grubby** 工具是否已安装在您的系统上。
- 验证 IBM Z 系统上是否已安装了 **zipl** 工具。

流程

- 添加参数：

```
# grubby --update-kernel=ALL --args="<NEW_PARAMETER>"
```

对于使用 GRUB 引导加载程序的系统，该命令会通过向该文件中的 **kernelopts** 变量添加一个新内核参数来更新 `/boot/grub2/grubenv` 文件。

- 在 IBM Z 上，更新引导菜单：

```
# zipl
```

- 删除参数：

```
# grubby --update-kernel=ALL --remove-args="<PARAMETER_TO_REMOVE>"
```

- 在 IBM Z 上，更新引导菜单：

```
# zipl
```



注意

新安装的内核继承之前配置的内核中的内核命令行参数。

其它资源

- [什么是内核命令行参数](#)
- [grubby\(8\) 和 zipl\(8\) 手册页](#)
- [grubby 工具](#)

4.4. 为单一引导条目更改内核命令行参数

对系统上单个引导条目的内核命令行参数进行更改。

先决条件

- 验证系统上已安装了 **grubby** 和 **zipl** 实用程序。

流程

- 添加参数：

```
# grubby --update-kernel=/boot/vmlinuz-$(uname -r) --args="<NEW_PARAMETER>"
```

- 在 IBM Z 上，更新引导菜单：

```
# zipl
```

- 删除参数：

```
# grubby --update-kernel=/boot/vmlinuz-$(uname -r) --remove-args="<PARAMETER_TO_REMOVE>"
```

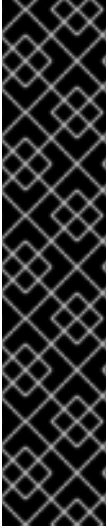
- 在 IBM Z 上，更新引导菜单：

```
# zipl
```



注意

在使用 **grub.cfg** 文件的系统上，默认情况下每个内核引导条目的 **options** 参数设置为 **kernelopts** 变量。此变量在 **/boot/grub2/grubenv** 配置文件中定义。



重要

在 GRUB2 系统上：

- 如果为所有引导条目修改了内核命令行参数，**grubby** 实用程序会更新 `/boot/grub2/grubenv` 文件中的 **kernelopts** 变量。
- 如果为单个引导条目修改了内核命令行参数，则扩展 **kernelopts** 变量，修改内核参数，结果值存储在相应的引导条目的 `/boot/loader/entries/<RELEVANT_KERNEL_BOOT_ENTRY.conf>` 文件中。

在 zipl 系统中：

- **grubby** 修改单个内核引导条目的内核命令行参数并将其存储在 `/boot/loader/entries/<ENTRY>.conf` 文件中。

其它资源

- [什么是内核命令行参数](#)
- **grubby(8)** 和 **zipl(8)** 手册页
- [grubby 工具](#)

4.5. 在引导时临时更改内核命令行参数

通过在单个引导过程中更改内核参数，对内核引导条目进行临时更改。

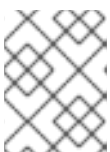


注意

这个过程只适用于单个引导，且不会使更改持久。

流程

1. 引导到 GRUB 2 引导菜单。
2. 选择您要启动的内核。
3. 按 **e** 键编辑内核参数。
4. 通过向下引动光标来找到内核命令行。在 64 位 IBM Power 系列和基于 x86-64 BIOS 的系统上内核命令行以 **linux** 开头，或在 UEFI 系统上以 **linuxefi** 开头。
5. 将光标移至行尾。



注意

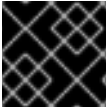
按 **Ctrl+a** 跳到行首，按 **Ctrl+e** 跳到行尾。在某些系统上，**Home** 和 **End** 键也可能正常工作。

6. 根据需要编辑内核参数。例如，要在紧急模式下运行系统，请在 **linux** 行末尾添加 **emergency** 参数：

```
linux ($root)/vmlinuz-4.18.0-348.12.2.el8_5.x86_64 root=/dev/mapper/rhel-root ro
crashkernel=auto resume=/dev/mapper/rhel-swap rd.lvm.lv=rhel/root rd.lvm.lv=rhel/swap
rhgb quiet emergency
```

要启用系统消息，请删除 **rhgb** 和 **quiet** 参数。

- 按 **Ctrl+x** 来使用选定的内核和修改的命令行参数启动。



重要

如果按 **Esc** 键离开命令行编辑，它将丢弃用户做的所有更改。

4.6. 配置 GRUB 设置以启用串行控制台连接

当您需要连接到无头服务器或嵌入式系统，且网络中断时，串行控制台非常有用。或者，当您需要避免安全规则，并获得不同系统上的登录访问权限时。

您需要配置一些默认的 GRUB 设置，以使用串行控制台连接。

先决条件

- 您有 root 权限。

流程

- 将下面两行添加到 **/etc/default/grub** 文件中：

```
GRUB_TERMINAL="serial"
GRUB_SERIAL_COMMAND="serial --speed=9600 --unit=0 --word=8 --parity=no --stop=1"
```

第一行将禁用图形终端。**GRUB_TERMINAL** 键覆盖 **GRUB_TERMINAL_INPUT** 和 **GRUB_TERMINAL_OUTPUT** 键的值。

第二行调整了波特率(**--speed**)，奇偶校验和其他值以适合您的环境和硬件。请注意，对于以下日志文件等任务，最好使用更高的波特率，如 115200。

- 更新 GRUB 配置文件。

- 在基于 BIOS 的机器上：

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

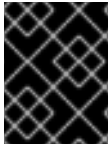
- 在基于 UEFI 的机器上：

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

- 重启系统以使更改生效。

第 5 章 在运行时配置内核参数

作为系统管理员，您可以修改 Red Hat Enterprise Linux 内核在运行时行为的很多方面。使用 **sysctl** 命令，并修改 **/etc/sysctl.d/** 和 **/proc/sys/** 目录中的配置文件来在运行时配置内核参数。



重要

在产品系统中配置内核参数需要仔细规划。未计划的更改可能会导致内核不稳定，需要重启系统。在更改任何内核值之前，验证您是否正在使用有效选项。

5.1. 什么是内核参数

内核参数是可在系统运行时调整的可调整值。不需要重启或重新编译内核就可以使更改生效。

可以通过以下方法处理内核参数：

- **sysctl** 命令
- 挂载于 **/proc/sys/** 目录的虚拟文件系统
- **/etc/sysctl.d/** 目录中的配置文件

Tunables 被内核子系统划分为不同的类。Red Hat Enterprise Linux 有以下可调整类：

表 5.1. sysctl 类表

可调整类	子系统
abi	执行域和个人
crypto	加密接口
debug	内核调试接口
dev	特定于设备的信息
fs	全局和特定文件系统的 tunables
内核	全局内核 tunables
net	网络 tunables
sunrpc	Sun 远程过程调用 (NFS)
user	用户命名空间限制
vm	调整和管理内存、缓冲和缓存

其它资源

- **sysctl(8)** 和 **sysctl.d(5)** 手册页

5.2. 使用 SYSCTL 临时配置内核参数

使用 `sysctl` 命令在运行时临时设置内核参数。命令也可用于列出和过滤可调项。

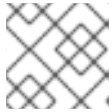
先决条件

- 根权限

流程

1. 列出所有参数及其值。

```
# sysctl -a
```



注意

`# sysctl -a` 命令显示内核参数，可在运行时和系统启动时调整。

2. 要临时配置一个参数，请输入：

```
# sysctl <TUNABLE_CLASS>.<PARAMETER>=<TARGET_VALUE>
```

上面的示例命令在系统运行时更改了参数值。更改将立即生效，无需重新启动。



注意

在系统重启后，所做的改变会返回到默认状态。

其它资源

- [sysctl \(8\) 手册页](#)
- [使用 sysctl 永久配置内核参数](#)
- [使用 /etc/sysctl.d/ 中的配置文件调整内核参数](#)

5.3. 使用 SYSCTL 永久配置内核参数

使用 `sysctl` 命令永久设置内核参数。

先决条件

- 根权限

流程

1. 列出所有参数。

```
# sysctl -a
```

该命令显示所有可在运行时配置的内核参数。

2. 永久配置一个参数：

```
# sysctl -w <TUNABLE_CLASS>.<PARAMETER>=<TARGET_VALUE> >> /etc/sysctl.conf
```

示例命令会更改可调值，并将其写入 `/etc/sysctl.conf` 文件，该文件会覆盖内核参数的默认值。更改会立即并永久生效，无需重启。

**注意**

要永久修改内核参数，您还可以手动更改 `/etc/sysctl.d/` 目录中的配置文件。

其它资源

- [sysctl \(8\) 和 sysctl.conf \(5\) 手册页](#)
- [使用 /etc/sysctl.d/ 中的配置文件调整内核参数](#)

5.4. 使用 /ETC/SYSCTL.D/ 中的配置文件调整内核参数

手动修改 `/etc/sysctl.d/` 目录中的配置文件，以永久设置内核参数。

先决条件

- 根权限

流程

1. 在 `/etc/sysctl.d/` 中创建一个新配置文件。

```
# vim /etc/sysctl.d/<some_file.conf>
```

2. 包括内核参数，一行一个。

```
<TUNABLE_CLASS>.<PARAMETER>=<TARGET_VALUE>
<TUNABLE_CLASS>.<PARAMETER>=<TARGET_VALUE>
```

3. 保存配置文件。
4. 重启机器以使更改生效。
 - 或者，要在不重启的情况下应用更改，请输入：

```
# sysctl -p /etc/sysctl.d/<some_file.conf>
```

该命令允许您从之前创建的配置文件中读取值。

其它资源

- [sysctl\(8\), sysctl.d\(5\) manual pages](#)

5.5. 通过 /PROC/SYS/ 临时配置内核参数

通过 `/proc/sys/` 虚拟文件系统目录中的文件临时设置内核参数。

先决条件

- 根权限

流程

1. 确定您要配置的内核参数。

```
# ls -l /proc/sys/<TUNABLE_CLASS>/
```

命令返回的可写入文件可以用来配置内核。具有只读权限的文件提供了对当前设置的反馈。

2. 为内核参数分配一个目标值。

```
# echo <TARGET_VALUE> > /proc/sys/<TUNABLE_CLASS>/<PARAMETER>
```

命令进行配置更改，这些更改将在系统重启后消失。

3. (可选) 验证新设置的内核参数的值。

```
# cat /proc/sys/<TUNABLE_CLASS>/<PARAMETER>
```

其它资源

- [使用 sysctl 永久配置内核参数](#)
- [使用 /etc/sysctl.d/ 中的配置文件调整内核参数](#)

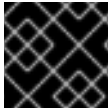
5.6. 其它资源

- [为 IBM DB2 调整 Red Hat Enterprise Linux](#)

第 6 章 对 GRUB 菜单进行临时更改

您可以修改 GRUB 菜单条目，或将参数传递给内核，这只适用于当前引导。在引导装载程序菜单中的所选菜单条目中，您可以：

- 按 **e** 键显示菜单条目编辑器界面。
- 按 **Esc** 键丢弃所有更改并重新载入标准菜单界面。
- 按 **c** 键来加载命令行界面。
- 输入所有相关 GRUB 命令，并通过按 **Enter** 键输入它们。
- 按 **Tab** 键，根据上下文完成命令。
- 按 **Ctrl+a** 组合键移动到行的开头。
- 按 **Ctrl+e** 组合键移动到行尾。



重要

以下流程提供了在单一引导过程中更改 GRUB 菜单的说明。

6.1. GRUB 简介

GRUB 代表 **GNU GRand Unified Bootloader**。有了 GRUB，您可以选择在系统引导时要载入的操作系统或内核。另外，您可以将参数传递给内核。

使用 GRUB 引导时，您可以使用菜单界面或命令行界面 (**GRUB 命令 shell**)。当您启动系统时，会显示菜单界面。

```

Red Hat Enterprise Linux (4.18.0-372.9.1.el8.x86_64) 8.6 (Ootpa)
Red Hat Enterprise Linux (0-rescue-67db13ba8cdb420794ef3ee0a8313205) 8.6→

Use the ↑ and ↓ keys to change the selection.
Press 'e' to edit the selected item, or 'c' for a command prompt.

```

您可以通过按 **c** 键切换到命令行界面。

```
Minimal BASH-like line editing is supported. For the first word,
TAB lists possible command completions. Anywhere else TAB lists
possible device or file completions. ESC at any time exits.
```

```
grub>
```

您可以输入 `exit` 并按 `Enter` 键来返回到菜单界面。

GRUB BLS 文件

引导装载程序菜单条目被定义为引导加载程序规格(BLS)文件。此文件格式管理置入目录中每个引导选项的引导装载程序配置，而不操作引导装载程序配置文件。`grubby` 工具可编辑这些 **BLS** 文件。

GRUB 配置文件

`/boot/grub2/grub.cfg` 配置文件不定义菜单条目。

其它资源

- [引导菜单](#)
- [引导装载程序规格介绍](#)

6.2. 引导装载程序规格介绍

BootLoader 规范 (BLS) 定义一个方案以及文件格式，以管理置入目录中每个引导选项的启动加载器配置，而无需操作启动加载器配置文件。与之前的方法不同，每个引导条目现在都由置入目录中的单独配置文件表示。置入目录扩展了其配置，无需编辑或重新生成配置文件。BLS 在引导菜单项中扩展了这个概念。

使用 BLS，您可以通过在目录中添加、删除或编辑单独的引导条目文件来管理启动加载器菜单选项。这使得内核安装过程在不同的构架中保持简单且一致。

`grubby` 工具是围绕 BLS 的精简打包程序脚本，它支持相同的 `grubby` 参数和选项。它运行 `dracut` 以创建初始 ramdisk 镜像。在这个版本中，核心引导装载程序配置文件是静态的，且在内核安装后不会修改。

这个前提在 RHEL 8 中尤其重要，因为并非所有架构都使用相同的引导装载程序。GRUB 用于大部分架构，如 64 位 ARM，但带有 Open Power Abstraction Layer (OPAL) 的 IBM Power Systems 的 little-endian 变体使用 **Petitboot**，IBM Z 架构使用 **zipl**。

其它资源

- [第 4.2 节 “了解引导条目”](#)
- [grubby \(8\) 手册页](#)

6.3. 引导至救援模式

救援模式提供了一个方便的用户环境，在此环境中，您可以在无法完成正常引导过程时修复您的系统。在救援模式下，系统会尝试挂载所有本地文件系统，并启动一些重要的系统服务。但是，它不会激活网络接口，或者允许更多的用户同时登录到系统。

流程

1. 在 GRUB 引导屏幕上，按 **e** 键进行编辑。
2. 在 **linux** 行末尾添加以下参数：

```
systemd.unit=rescue.target
```

```
load_video
set gfx_payload=keep
insmod gzio
linux ($root)/vmlinuz-4.18.0-372.9.1.el8.x86_64 root=/dev/mapper/rhel-root ro \
crashkernel=auto resume=/dev/mapper/rhel-swap rd.lvm.lv=rhel/root rd.lvm.lv=rh\
el/swap rhgb quiet zswap.enabled=0 systemd.unit=rescue.target
initrd ($root)/initramfs-4.18.0-372.9.1.el8.x86_64.img $tuned_initrd
```

3. 按 **Ctrl+x** 引导至救援模式。

```
You are in rescue mode. After logging in, type "journalctl -xb" to view
system logs, "systemctl reboot" to reboot, "systemctl default" or "exit"
to boot into default mode.
Give root password for maintenance
(or press Control-D to continue):
```

6.4. 引导至紧急模式

紧急模式提供尽可能小的环境，在此环境中，您可以在系统无法进入到救援模式时修复您的系统。

在紧急模式下，系统：

- 仅为读取挂载 **root** 文件系统
- 启动几个基本服务

但是，系统 **不会**：

- 尝试挂载任何其他本地文件系统
- 激活网络接口

流程

1. 在 GRUB 引导屏幕上，按 **e** 键进行编辑。
2. 在 **linux** 行末尾添加以下参数：

```
systemd.unit=emergency.target
```

```
load_video
set gfx_payload=keep
insmod gzio
linux ($root)/vmlinuz-4.18.0-372.9.1.el8.x86_64 root=/dev/mapper/rhel-root ro \
crashkernel=auto resume=/dev/mapper/rhel-swap rd.lvm.lv=rhel/root rd.lvm.lv=rh\
el/swap rhgb quiet zswap.enabled=0 systemd.unit=emergency.target
initrd ($root)/initramfs-4.18.0-372.9.1.el8.x86_64.img $tuned_initrd
```

- 按 **Ctrl+x** 引导至紧急模式。

```
You are in emergency mode. After logging in, type "journalctl -xb" to view
system logs, "systemctl reboot" to reboot, "systemctl default" or "exit"
to boot into default mode.
Give root password for maintenance
(or press Control-D to continue): _
```

6.5. 引导至 DEBUG SHELL

systemd debug shell 在启动过程早期提供一个 shell。在 debug shell 中，您可以使用 **systemctl** 命令，如 **systemctl list-jobs** 和 **systemctl list-units**，搜索与引导问题相关的 **systemd** 的原因。

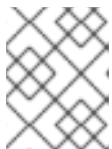
流程

- 在 GRUB 引导屏幕上，按 **e** 键进行编辑。
- 在 **linux** 行末尾添加以下参数：

```
systemd.debug-shell
```

```
load_video
set gfx_payload=keep
insmod gzio
linux ($root)/vmlinuz-4.18.0-372.9.1.el8.x86_64 root=/dev/mapper/rhel-root ro \
crashkernel=auto resume=/dev/mapper/rhel-swap rd.lvm.lv=rhel/root rd.lvm.lv=rh\
el/swap rhgb quiet systemd.debug-shell
initrd ($root)/initramfs-4.18.0-372.9.1.el8.x86_64.img $tuned_initrd
```

- (可选) 添加 **debug** 选项。



注意

向内核命令行添加 **debug** 选项会增加日志消息的数量。对于 **systemd**，内核命令行选项 **debug** 现在是 **systemd.log_level=debug** 的快捷方式。

- 按 **Ctrl+x** 启动到 debug shell。



警告

永久启用 debug shell 是一个安全风险，因为不需要进行身份验证就可以使用它。当调试会话结束时禁用它。

6.6. 连接到 DEBUG SHELL

在启动过程中，**systemd-debug-generator** 在 TTY9 上配置 debug shell。

先决条件

- 您已成功引导至 debug shell。请参阅 [引导至 debug shell](#)。

流程

1. 按 **Ctrl+Alt+F9** 连接到 debug shell。
如果您使用虚拟机，发送此组合键需要虚拟化应用程序的支持。例如，如果使用 **虚拟机管理器**，请从菜单中选择 **Send Key → Ctrl+Alt+F9**。
2. debug shell 不需要身份验证，因此您可以在 TTY9 上看到类似如下的提示：

```
sh-4.4#
```

验证步骤

- 按照如下所示输入命令：

```
sh-4.4# systemctl status $$
```

```
sh-4.4# systemctl status $$
■ debug-shell.service - Early root shell on /dev/tty9 FOR DEBUGGING ONLY
   Loaded: loaded (/usr/lib/systemd/system/debug-shell.service; enabled-runtime; vendor preset: disabled)
   Active: active (running) since Tue 2022-08-02 08:40:09 EDT; 45s ago
     Docs: man:systemd-debug-generator(8)
    Main PID: 735 (sh)
      Tasks: 3 (limit: 17257)
     Memory: 2.3M
    CGroup: /system.slice/debug-shell.service
            └─ 735 /bin/sh
               └─ 2092 systemctl status 735
                  └─ 2093 less

sh-4.4#
```

- 要返回到默认 shell，如果引导成功，请按 **Ctrl+Alt+F1**。

其它资源

- [systemd-debug-generator \(8\) 手册页](#)

6.7. 使用安装盘重置 ROOT 密码

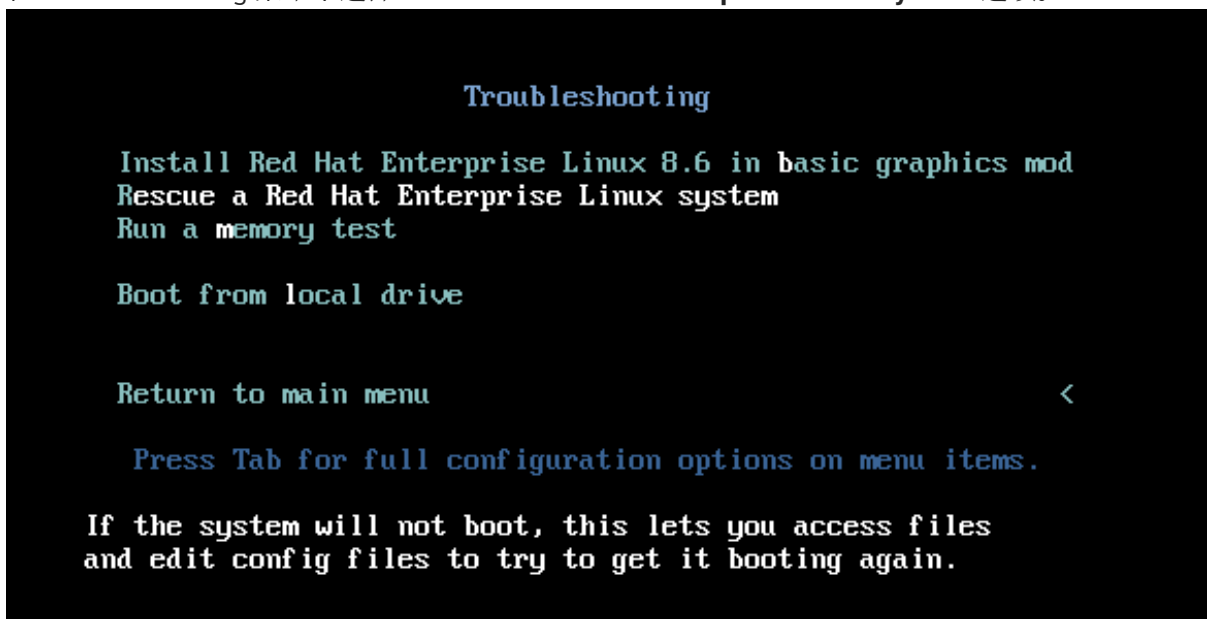
如果您忘记或丢失了 **root** 密码，您可以重置它。

流程

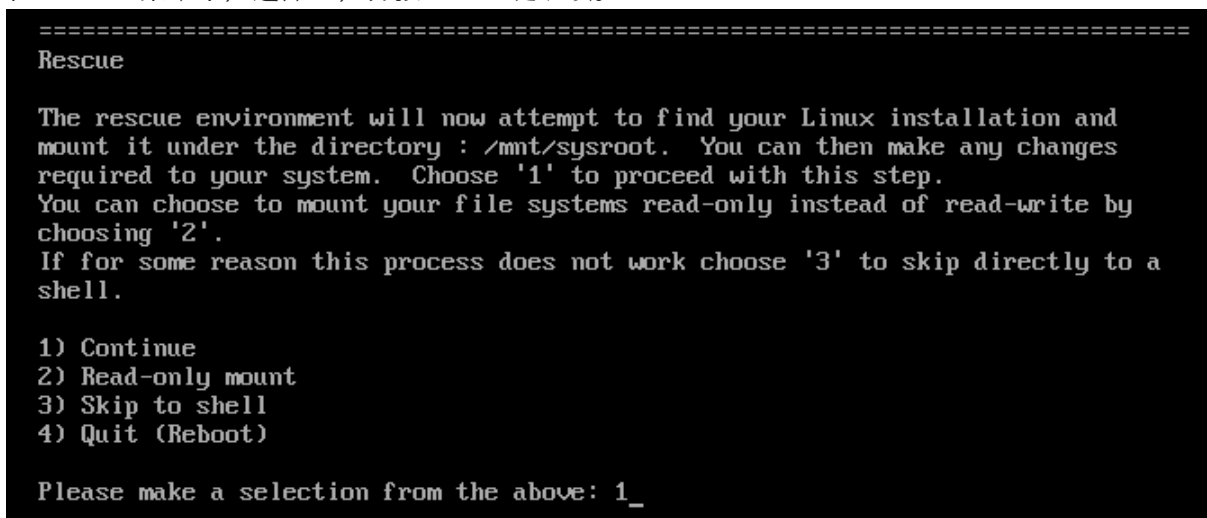
1. 从安装源引导主机。
2. 在安装介质的引导菜单中，选择 **Troubleshooting** 选项。



3. 在 Troubleshooting 菜单中选择 **Rescue a Red Hat Enterprise Linux system** 选项。



4. 在 Rescue 菜单中，选择 **1**，并按 **Enter** 键继续。



5. 按如下所示更改文件系统 **root** :

```
sh-4.4# chroot /mnt/sysimage
```



```

When finished, please exit from the shell and your system will reboot.
Please press ENTER to get a shell:
sh-4.4# chroot /mnt/sysimage
bash-4.4#

[anaconda]1:main* 2:shell 3:log 4:storage-log 5:program-log

```

6. 输入 `passwd` 命令，并按照命令行上显示的指令更改 `root` 密码。

```

bash-4.4# passwd
Changing password for user root.
New password:
Retype new password:
passwd: all authentication tokens updated successfully.
bash-4.4#

[anaconda]1:main* 2:shell 3:log 4:storage-log 5:program-log

```

7. 删除 `autorelabel` 文件，以防止耗时的磁盘的 SELinux 重新标记：

```

sh-4.4# rm -f /.autorelabel

```

8. 输入 `exit` 命令退出 `chroot` 环境。
9. 再次输入 `exit` 命令，以恢复初始化并完成系统启动。

6.8. 使用 RD.BREAK 重置 ROOT 密码

如果您忘记或丢失了 `root` 密码，您可以重置它。

流程

1. 启动系统，并在 GRUB 引导屏幕上按 `e` 键进行编辑。
2. 在 `linux` 行末尾添加 `rd.break` 参数：

```

load_video
set gfx_payload=keep
insmod gzio
linux ($root)/vmlinuz-4.18.0-372.9.1.el8.x86_64 root=/dev/mapper/rhel-root ro \
crashkernel=auto resume=/dev/mapper/rhel-swap rd.lvm.lv=rhel/root rd.lvm.lv=rh\
el/swap rhgb quiet rd.break
initrd ($root)/initramfs-4.18.0-372.9.1.el8.x86_64.img $tuned_initrd

```

3. 按 `Ctrl+x` 使用更改的参数引导系统。

```
Generating "/run/initramfs/rdsosreport.txt"

Entering emergency mode. Exit the shell to continue.
Type "journalctl" to view system logs.
You might want to save "/run/initramfs/rdsosreport.txt" to a USB stick or /boot
after mounting them and attach it to a bug report.

switch_root:/# _
```

4. 将文件系统重新挂载为可写。

```
switch_root:/# mount -o remount,rw /sysroot
```

5. 更改文件系统的 **root**。

```
switch_root:/# chroot /sysroot
```

6. 输入 **passwd** 命令，并按照命令行上显示的说明进行操作。

```
bash-4.4# passwd
Changing password for user root.
New password:
Retype new password:
passwd: all authentication tokens updated successfully.
bash-4.4#

[anaconda]1:main* 2:shell 3:log 4:storage-log 5:program-log
```

7. 在下次系统引导时重新标记所有文件。

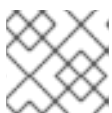
```
sh-4.4# touch /.autorelabel
```

8. 重新将文件系统挂载为 **只读**：

```
sh-4.4# mount -o remount,ro /
```

9. 输入 **exit** 命令退出 **chroot** 环境。

10. 再次输入 **exit** 命令，以恢复初始化并完成系统启动。



注意

SELinux 重新标记过程可能需要很长时间。系统会在进程完成后自动重启。

提示

您可以通过添加 **enforcing=0** 选项来省略耗时的 SELinux 重新标记过程。

流程

1. 当在 **linux** 行末尾添加 **rd.break** 参数时，请附加 **enforcing=0**。

```
rd.break enforcing=0
```

2. 恢复 **/etc/shadow** 文件的 SELinux 安全上下文。

```
# restorecon /etc/shadow
```

3. 重新打开 SELinux 策略强制，并确认它是否开启。

```
# setenforce 1
# getenforce
Enforcing
```

请注意，如果您在第 3 步中添加了 **enforcing=0** 选项，您可以在第 8 步中省略输入 **touch /.autorelabel** 命令。

6.9. 其它资源

- **/usr/share/doc/grub2-common** 目录。
- **info grub2** 命令。

第 7 章 对 GRUB 引导装载程序进行永久性更改

使用 **grubby** 工具在 GRUB 中进行永久更改。

7.1. 先决条件

- 您已在您的系统上成功安装了 RHEL。
- 您有 root 权限。

7.2. 列出默认的内核

通过列出默认内核，您可以找到默认内核的文件名和索引号，以对 GRUB 引导装载程序进行永久更改。

流程

- 要找到默认内核的文件名，请输入：

```
# grubby --default-kernel
/boot/vmlinuz-4.18.0-372.9.1.el8.x86_64
```

- 要找到默认内核的索引号，请输入：

```
# grubby --default-index
0
```

7.3. 查看内核的 GRUB 菜单条目

您可以列出所有内核菜单条目，或者查看特定内核的 GRUB 菜单条目。

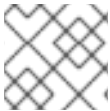
流程

- 要列出所有内核菜单条目，请输入：

```
# grubby --info=ALL
index=0
kernel="/boot/vmlinuz-4.18.0-372.9.1.el8.x86_64"
args="ro crashkernel=auto resume=/dev/mapper/rhel-swap rd.lvm.lv=rhel/root
rd.lvm.lv=rhel/swap rhgb quiet $tuned_params zswap.enabled=1"
root="/dev/mapper/rhel-root"
initrd="/boot/initramfs-4.18.0-372.9.1.el8.x86_64.img $tuned_initrd"
title="Red Hat Enterprise Linux (4.18.0-372.9.1.el8.x86_64) 8.6 (Ootpa)"
id="67db13ba8cdb420794ef3ee0a8313205-4.18.0-372.9.1.el8.x86_64"
index=1
kernel="/boot/vmlinuz-0-rescue-67db13ba8cdb420794ef3ee0a8313205"
args="ro crashkernel=auto resume=/dev/mapper/rhel-swap rd.lvm.lv=rhel/root
rd.lvm.lv=rhel/swap rhgb quiet"
root="/dev/mapper/rhel-root"
initrd="/boot/initramfs-0-rescue-67db13ba8cdb420794ef3ee0a8313205.img"
title="Red Hat Enterprise Linux (0-rescue-67db13ba8cdb420794ef3ee0a8313205) 8.6
(Ootpa)"
id="67db13ba8cdb420794ef3ee0a8313205-0-rescue"
```

- 要查看特定内核的 GRUB 菜单条目，请输入：

```
# grubby --info /boot/vmlinuz-4.18.0-372.9.1.el8.x86_64
grubby --info /boot/vmlinuz-4.18.0-372.9.1.el8.x86_64
index=0
kernel="/boot/vmlinuz-4.18.0-372.9.1.el8.x86_64"
args="ro crashkernel=auto resume=/dev/mapper/rhel-swap rd.lvm.lv=rhel/root
rd.lvm.lv=rhel/swap rhgb quiet $tuned_params zswap.enabled=1"
root="/dev/mapper/rhel-root"
initrd="/boot/initramfs-4.18.0-372.9.1.el8.x86_64.img $tuned_initrd"
title="Red Hat Enterprise Linux (4.18.0-372.9.1.el8.x86_64) 8.6 (Ootpa)"
id="67db13ba8cdb420794ef3ee0a8313205-4.18.0-372.9.1.el8.x86_64"
```



注意

尝试 tab 补全来查看 `/boot` 目录中的可用内核。

7.4. 编辑内核参数

您可以更改现有内核参数中的值。例如，您可以更改虚拟控制台（屏幕）字体和大小。

流程

- 将虚拟控制台字体更改为 `latarcyrheb-sun`，大小为 `32`。

```
# grubby --args=vconsole.font=latarcyrheb-sun32 --update-kernel /boot/vmlinuz-
4.18.0-372.9.1.el8.x86_64
```

7.5. 在 GRUB 菜单条目中添加和删除参数

您可以从 GRUB 菜单添加、删除或同时添加或删除参数。

流程

- 要向 GRUB 菜单条目中添加参数，请使用 `--update-kernel` 选项和 `--args`。例如，以下命令添加了一个串行控制台：

```
# grubby --args=console=ttyS0,115200 --update-kernel /boot/vmlinuz-4.18.0-
372.9.1.el8.x86_64
```

控制台参数附加到行尾，新控制台将优先于任何其他配置的控制台。

- 要从 GRUB 菜单条目中删除参数，请使用 `--update-kernel` 选项和 `--remove-args`。例如：

```
# grubby --remove-args="rhgb quiet" --update-kernel /boot/vmlinuz-4.18.0-
372.9.1.el8.x86_64
```

这个命令会删除红帽图形引导参数，并启用日志消息，这是详细模式。

- 要同时添加和删除参数，请输入：

```
# grubby --remove-args="rhgb quiet" --args=console=ttyS0,115200 --update-kernel
/boot/vmlinuz-4.18.0-372.9.1.el8.x86_64
```

验证步骤

- 要查看您所做的永久更改，请输入：

```
# grubby --info /boot/vmlinuz-4.18.0-372.9.1.el8.x86_64
index=0
kernel="/boot/vmlinuz-4.18.0-372.9.1.el8.x86_64"
args="ro crashkernel=auto resume=/dev/mapper/rhel-swap rd.lvm.lv=rhel/root
rd.lvm.lv=rhel/swap $tuned_params zswap.enabled=1 console=ttyS0,115200"
root="/dev/mapper/rhel-root"
initrd="/boot/initramfs-4.18.0-372.9.1.el8.x86_64.img $tuned_initrd"
title="Red Hat Enterprise Linux (4.18.0-372.9.1.el8.x86_64) 8.6 (Ootpa)"
id="67db13ba8cdb420794ef3ee0a8313205-4.18.0-372.9.1.el8.x86_64"
```

7.6. 添加一个新的引导条目

您可以向引导装载程序菜单条目中添加一个新的引导条目。

流程

1. 将来自默认内核的所有内核参数复制到这个新的内核条目。

```
# grubby --add-kernel=new_kernel --title="entry_title" --initrd="new_initrd" --copy-default
```

2. 获取可用的引导条目的列表。

```
# ls -l /boot/loader/entries/*
-rw-r--r--. 1 root root 408 May 27 06:18
/boot/loader/entries/67db13ba8cdb420794ef3ee0a8313205-0-rescue.conf
-rw-r--r--. 1 root root 536 Jun 30 07:53
/boot/loader/entries/67db13ba8cdb420794ef3ee0a8313205-4.18.0-372.9.1.el8.x86_64.conf
-rw-r--r-- 1 root root 336 Aug 15 15:12
/boot/loader/entries/d88fa2c7ff574ae782ec8c4288de4e85-4.18.0-193.el8.x86_64.conf
```

3. 创建一个新的引导条目。例如，对于 `4.18.0-193.el8.x86_64` 内核，请按如下所示运行命令：

```
# grubby --grub2 --add-kernel=/boot/vmlinuz-4.18.0-193.el8.x86_64 --title="Red Hat Enterprise 8 Test" --initrd=/boot/initramfs-4.18.0-193.el8.x86_64.img --copy-default
```

验证

- 验证新添加的引导条目是否已列在可用的引导条目中。

```
# ls -l /boot/loader/entries/*
-rw-r--r--. 1 root root 408 May 27 06:18
/boot/loader/entries/67db13ba8cdb420794ef3ee0a8313205-0-rescue.conf
-rw-r--r--. 1 root root 536 Jun 30 07:53
/boot/loader/entries/67db13ba8cdb420794ef3ee0a8313205-4.18.0-372.9.1.el8.x86_64.conf
-rw-r--r-- 1 root root 287 Aug 16 15:17
/boot/loader/entries/d88fa2c7ff574ae782ec8c4288de4e85-4.18.0-
```

193.el8.x86_64.0~custom.conf

-rw-r--r-- 1 root root 287 Aug 16 15:29

/boot/loader/entries/d88fa2c7ff574ae782ec8c4288de4e85-4.18.0-193.el8.x86_64.conf

7.7. 使用 GRUBBY 更改默认引导条目

使用 **grubby** 工具，您可以更改默认引导条目。

流程

- 要在指定为默认内核的内核中进行持久更改，请输入：

```
# grubby --set-default /boot/vmlinuz-4.18.0-372.9.1.el8.x86_64
The default is /boot/loader/entries/67db13ba8cdb420794ef3ee0a8313205-4.18.0-372.9.1.el8.x86_64.conf with index 0 and kernel /boot/vmlinuz-4.18.0-372.9.1.el8.x86_64
```

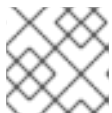
7.8. 使用同样的参数更新所有内核菜单

您可以向所有内核菜单条目中添加相同的内核引导参数。

流程

- 要向所有内核菜单条目中添加相同的内核引导参数，请附加 **--update-kernel=ALL** 参数。例如，这个命令向所有内核添加一个串行控制台：

```
# grubby --update-kernel=ALL --args=console=ttyS0,115200
```



注意

--update-kernel 参数还接受 **DEFAULT** 或以逗号分隔的内核索引号列表。

7.9. 为当前和将来的内核更改默认内核选项

通过使用 **kernelopts** 变量，您可以为当前和将来的内核更改默认内核选项。

流程

1. 列出 **kernelopts** 变量中的内核参数：

```
# grub2-editenv - list | grep kernelopts
kernelopts=root=/dev/mapper/rhel-root ro crashkernel=auto resume=/dev/mapper/rhel-swap
rd.lvm.lv=rhel/root rd.lvm.lv=rhel/swap rhgb quiet
```

2. 对内核命令行参数进行更改。您可以添加、删除或修改参数。例如，要添加 **debug** 参数，请输入：

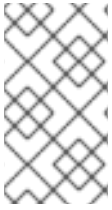
```
# grub2-editenv - set "$(grub2-editenv - list | grep kernelopts) <debug>"
```

3. 可选：验证新添加的参数是否已添加到 **kernelopts** 中：

```
# grub2-editenv - list | grep kernelopts
```

```
kernelopts=root=/dev/mapper/rhel-root ro crashkernel=auto resume=/dev/mapper/rhel-swap  
rd.lvm.lv=rhel/root rd.lvm.lv=rhel/swap rhgb quiet debug
```

4. 重启系统以使更改生效。



注意

另一种方法，您可以使用 **grubby** 命令将参数传递给当前和将来的内核：

```
# grubby --update-kernel ALL --args="<PARAMETER>"
```

7.10. 其它资源

- `/usr/share/doc/grub2-common` 目录。
- `info grub2` 命令。

第 8 章 构建一个自定义引导菜单

您可以构建一个包含特定条目的引导菜单或更改条目的顺序。对于这样的任务，您可以使用 GRUB、**grubby** 和引导启动程序规范(BLS)文件。

以下章节提供了有关使用 GRUB 和 **grubby** 对引导菜单进行基本自定义的信息。

8.1. GRUB 配置文件

了解基于 BIOS 的机器上的 `/boot/grub2/grub.cfg` 的引导装载程序配置文件，以及基于 UEFI 的机器上的 `/boot/efi/EFI/redhat/grub.cfg`。

GRUB 脚本会搜索用户的计算机，并根据脚本发现的操作系统的类型构建引导菜单。要反映最新的系统引导选项，在内核更新或添加新内核时会自动重建引导菜单。

GRUB 使用一系列脚本来构建菜单；这些脚本位于 `/etc/grub.d/` 目录中。包括以下文件：

- **00_header**，它从 `/etc/default/grub` 文件中加载 GRUB 设置。
- **01_users**，它从 `user.cfg` 文件中读取 root 密码。
- **10_Linux**，在 Red Hat Enterprise Linux 的默认分区中查找内核。
- **30_os-prober**，它为在其他分区上发现的操作系统构建条目。
- **40_custom**，一个模板，可用于创建额外的菜单条目。

GRUB 以字母顺序从 `/etc/grub.d/` 目录中读取脚本，因此您可以重命名它们以更改特定菜单条目的引导顺序。

8.2. 隐藏可引导的内核的列表

您可以防止 GRUB 在系统启动时显示可引导内核的列表。

流程

1. 在 `/etc/default/grub` 文件中设置 **GRUB_TIMEOUT_STYLE** 选项，如下所示：

```
GRUB_TIMEOUT_STYLE=hidden
```

2. 重建 **grub.cfg** 文件以使更改生效。

- 在基于 BIOS 的机器上，输入：

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

- 在基于 UEFI 的机器上，输入：

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

3. 按 **Esc** 键，以显示启动时可引导内核的列表。



重要

不要在 `/etc/default/grub` 文件中将 `GRUB_TIMEOUT` 设为 `0` 来隐藏可引导内核的列表。有了这个设置，系统总是在默认菜单条目上立即引导，如果默认内核无法引导，则无法引导任何以前的内核。

8.3. 使用 GRUB 配置文件更改默认引导条目

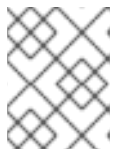
您可以指定默认内核软件包类型，从而更改默认引导条目。

流程

1. 通过将其索引传给 `grub2-set-default` 命令，来指定默认载入哪个操作系统或内核。例如：

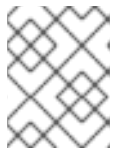
```
# grubby --set-default-index=1
The default is /boot/loader/entries/d5151aa93c444ac89e78347a1504d6c6-4.18.0-348.el8.x86_64.conf with index 1 and kernel /boot/vmlinuz-4.18.0-348.el8.x86_64
```

GRUB 支持将数字值用作 `/boot/grub2/grubenv` 中 `saved_entry` 指令的密钥，以更改载入操作系统的默认顺序。



注意

索引计数从零开始；因此，在上例中，GRUB 加载第二个条目。在下一个安装的内核中，索引值将被覆盖。



注意

您还可以使用 `grubby` 查找内核的索引。如需更多信息，请参阅 [查看内核的 GRUB 菜单条目](#)。

2. 可选：强制系统总是使用特定的菜单条目：

- a. 列出可用的菜单条目：

```
# grubby --info=ALL
```

- b. 使用菜单条目名称或列表中的菜单条目的位置号来作为 `/etc/default/grub` 文件中 `GRUB_DEFAULT` 指令的密钥。例如：

```
GRUB_DEFAULT=example-gnu-linux
```

3. 重建 `grub.cfg` 文件以使更改生效。

- 在基于 BIOS 的机器上，输入：

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

- 在基于 UEFI 的机器上：

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

第 9 章 重新安装 GRUB

您可以重新安装 GRUB 引导装载程序来修复某些问题，通常是这些问题是由于不正确的 GRUB 安装、缺少文件或损坏的系统而造成的。您可以通过恢复缺少的文件并更新引导信息来解决这个问题。

重新安装 GRUB 的原因：

- 升级 GRUB 引导装载程序软件包。
- 将引导信息添加到另一个驱动器中。
- 用户需要 GRUB 引导装载程序来控制安装的操作系统。但是，一些操作系统是使用自己的引导装载程序安装的，重新安装 GRUB 将控制权返回所需的操作系统。



注意

只有在文件没有损坏时，GRUB 才能恢复这些文件。

9.1. 在基于 BIOS 的机器上重新安装 GRUB

您可以在基于 BIOS 的系统上重新安装 GRUB 引导装载程序。更新 GRUB 软件包后，始终重新安装 GRUB。



重要

这会覆盖现有的 GRUB，以安装新的 GRUB。确定系统在安装过程中不会导致数据损坏或引导崩溃。

流程

1. 在安装了 GRUB 的设备上重新安装它。例如：如果 **sda** 是您的设备：

```
# grub2-install /dev/sda
```

2. 重启您的系统以使更改生效：

```
# reboot
```

其它资源

- [grub-install \(1\) 手册页](#)

9.2. 在基于 UEFI 的机器上重新安装 GRUB

您可以在基于 UEFI 的系统上重新安装 GRUB 引导装载程序。



重要

确定系统在安装过程中不会导致数据损坏或引导崩溃。

流程

1. 重新安装 **grub2-efi** 和 **shim** 引导装载程序文件：

```
# yum reinstall grub2-efi shim
```

2. 重启您的系统以使更改生效：

```
# reboot
```

9.3. 在 IBM POWER 机器上重新安装 GRUB

您可以在 IBM Power 系统的 Power PC Reference Platform(PReP)引导分区上重新安装 GRUB 引导装载程序。更新 GRUB 软件包后，始终重新安装 GRUB。



重要

这会覆盖现有的 GRUB，以安装新的 GRUB。确定系统在安装过程中不会导致数据损坏或引导崩溃。

流程

1. 确定存储 GRUB 的磁盘分区：

```
# bootlist -m normal -o  
sda1
```

2. 在磁盘分区上重新安装 GRUB：

```
# grub2-install partition
```

使用您在上一步中找到的 GRUB 分区（如 `/dev/sda1`）替换 *partition*。

3. 重启您的系统以使更改生效：

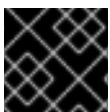
```
# reboot
```

其它资源

- [grub-install \(1\) 手册页](#)

9.4. 重置 GRUB

重置 GRUB 会完全删除所有的 GRUB 配置文件和系统设置，并重新安装启动加载程序。您可以将所有配置重置为默认值，从而修复由文件损坏和不正确的配置导致的故障。



重要

以下流程将删除用户所做的所有自定义。

流程

1. 删除配置文件。

■

```
# rm /etc/grub.d/*  
# rm /etc/sysconfig/grub
```

2. 重新安装软件包。

- 在基于 BIOS 的机器上，输入：

```
# yum reinstall grub2-tools
```

- 在基于 UEFI 的机器上，输入：

```
# yum reinstall grub2-efi shim grub2-tools
```

3. 重建 `grub.cfg` 文件以使更改生效。

- 在基于 BIOS 的机器上，输入：

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

- 在基于 UEFI 的机器上，输入：

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

4. 按照 [重新安装 GRUB](#) 流程来在 `/boot/` 分区上恢复 GRUB。

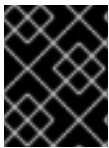
第 10 章 使用密码保护 GRUB

您可以通过两种方式使用密码保护 GRUB：

- 修改菜单条目需要密码，但引导现有菜单条目不需要。
- 修改菜单条目以及引导现有菜单条目时需要密码。

10.1. 仅为修改菜单条目设置密码保护

您可以配置 GRUB 以支持用于修改 GRUB 菜单条目的密码验证。这个过程会创建一个 `/boot/grub2/user.cfg` 文件，其包含哈希格式的密码。



重要

使用 `grub2-setpassword` 命令设置密码可防止菜单条目未经授权修改，但不会防止未经授权的启动。

流程

1. 以 root 用户身份发出 `grub2-setpassword` 命令。

```
# grub2-setpassword
```

2. 输入用户的密码，然后按 **Enter** 键确认密码。

```
Enter password:
Confirm the password:
```



注意

root 用户在 `/boot/grub2/grub.cfg` 文件中定义，并修改了密码。因此，在启动过程中修改引导条目需要 root 用户名和密码。

10.2. 对修改和引导菜单条目设置密码保护

您可以配置 GRUB 以防止菜单条目未经授权修改以及未经授权启动。



警告

如果您忘记了 GRUB 密码，您将无法引导您重新配置的条目。

流程

1. 打开您要修改 `/boot/loader/entries/` 目录中的引导条目的引导加载程序规范(BLS)文件。
2. 找到以 `grub_users` 开头的行。此参数向 `menuentry` 传递额外的参数。

3. 将 **grub_users** 属性设置为除超级用户之外的、允许启动条目的用户名，默认情况下，此用户为 **root**。下面是一个配置文件示例：

```
title Red Hat Enterprise Linux (4.18.0-221.el8.x86_64) 8.3
(Ootpa)
version 4.18.0-221.el8.x86_64
linux /vmlinuz-4.18.0-221.el8.x86_64
initrd /initramfs-4.18.0-221.el8.x86_64.img $tuned_initrd
options $kernelopts $tuned_params
id rhel-20200625210904-4.18.0-221.el8.x86_64
grub_users root
grub_arg --unrestricted
grub_class kernel
```

4. 保存并关闭 **BLS** 文件。



注意

如果要防止所有菜单条目引导，您可以直接设置 **grub_users** 属性。例如，如果用户是 **root**：

```
# grub2-editenv - set grub_users="root"
```

第 11 章 在虚拟环境中保留内核 PANIC 参数

在 RHEL 8 中配置虚拟机时，您不应该启用 `softlockup_panic` 和 `nmi_watchdog` 内核参数，因为虚拟机可能遭受虚假的软锁定。这不需要一个内核 panic。

在以下部分中找到此建议背后的原因。

11.1. 什么是软锁定

当任务在不重新调度的情况下在 CPU 上的内核空间中执行时，软锁定通常是由程序错误造成的。该任务也不允许任何其他任务在特定 CPU 上执行。因此，用户通过系统控制台会显示警告信息。这个问题也被称为软锁定触发。

其它资源

- [什么是 CPU 软锁定？](#)

11.2. 控制内核 PANIC 的参数

可设置以下内核参数来控制当检测到软锁定时的系统行为。

`softlockup_panic`

控制当检测到软锁定时内核是否 panic。

类型	值	效果
整数	0	内核在软锁定时不 panic
整数	1	软锁定中的内核 panics

默认情况下，在 RHEL8 上，这个值为 0。

系统需要首先检测硬锁定才能 panic。检测由 `nmi_watchdog` 参数控制。

`nmi_watchdog`

控制锁定检测机制 (`watchdogs`) 是否处于活动状态。这个参数是整数类型。

值	效果
0	禁用锁定检测器
1	启用锁定检测器

硬锁定检测器会监控每个 CPU 是否有响应中断的能力。

`watchdog_thresh`

控制 `watchdog` `hrtimer`、NMI 事件和软/硬锁定阈值的频率。

默认阈值	软锁定阈值
10 秒	$2 * \text{watchdog_thresh}$

将此参数设置为 0 可禁用锁定检测。

其它资源

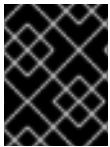
- [软锁定检测器和硬锁定检测器](#)
- [内核 sysctl](#)

11.3. 在虚拟环境中伪装软锁定

软锁定在物理主机上触发，如[什么是软锁定](#)中所述，这通常代表内核或硬件漏洞。在虚拟环境中的客户端操作系统中会出现同样的问题。

主机的高工作负载或者某些特定资源（如内存）的高竞争，通常会导致错误的软锁定触发。这是因为主机可能会调度出客户端 CPU 的时间超过 20 秒。然后，当客户机 CPU 再次被调度到在主机上运行时，它会遇到一个[时间跳转 \(time jump\)](#)，这会触发到时计时器。计时器还包括 watchdog **hrtimer**，因此可以在客户机 CPU 上报告软锁定。

因为虚拟化环境中的软锁定可能是假的，所以您不应该启用在客户端 CPU 上报告软锁定时导致系统 panic 的内核参数。



重要

若要了解客户机中的软锁定，必须了解，主机会作为一个任务调度客户机，客户机然后会调度自己的任务。

其它资源

- [什么是软锁定](#)
- [虚拟机组件及其交互](#)
- [虚拟机报告了"BUG：软锁定"](#)

第 12 章 为数据库服务器调整内核参数

有一组不同的内核参数可能会影响特定数据库应用程序的性能。为了保护数据库服务器和数据库的高效操作，请相应地配置对应的内核参数。

12.1. 介绍

数据库服务器是一种提供数据库管理系统 (DBMS) 功能的服务。DBMS 提供数据库管理的实用程序，并与最终用户、应用程序和数据库交互。

Red Hat Enterprise Linux 8 提供以下数据库管理系统：

- MariaDB 10.3
- MariaDB 10.5 - 从 RHEL 8.4 开始提供
- MariaDB 10.11 - available since RHEL 8.10
- MySQL 8.0
- PostgreSQL 10
- PostgreSQL 9.6
- PostgreSQL 12 - 从 RHEL 8.1.1 开始提供
- PostgreSQL 13 - 从 RHEL 8.4 开始提供
- PostgreSQL 15 - 从 RHEL 8.8 开始提供
- PostgreSQL 16 - 从 RHEL 8.10 开始可用

12.2. 影响数据库应用程序性能的参数

以下内核参数会影响数据库应用程序的性能。

fs.aio-max-nr

定义系统可在服务器中处理的异步 I/O 操作的最大数目。



注意

增加 **fs.aio-max-nr** 参数不会在增加 aio 限制外产生任何变化。

fs.file-max

定义系统在任何实例上支持的最大文件句柄数（临时文件名或者分配给打开文件的 ID）。内核会在应用程序请求文件句柄时动态分配文件。但是，当应用程序发布这些文件时，内核不会释放这些文件句柄。相反，内核会回收这些文件的句柄。这意味着，分配的文件句柄总数将随着时间增加，即使当前使用的文件句柄的数量可能较低。

kernel.shmall

定义可用于系统范围的共享内存页面总数。要使用整个主内存，**kernel.shmall** 参数的值应当为主内存大小总计。

kernel.shmmax

定义 Linux 进程在其虚拟地址空间中可分配的单个共享内存段的最大字节大小。

kernel.shmni

定义数据库服务器能够处理的共享内存段的最大数量。

net.ipv4.ip_local_port_range

定义系统可用于希望在不特定端口号的情况下连接到数据库服务器的程序的端口范围。

net.core.rmem_default

通过传输控制协议 (TCP) 定义默认接收套接字内存。

net.core.rmem_max

通过传输控制协议 (TCP) 定义最大接收套接字内存。

net.core.wmem_default

通过传输控制协议 (TCP) 定义默认发送套接字内存。

net.core.wmem_max

通过传输控制协议 (TCP) 定义最大发送套接字内存。

vm.dirty_bytes / vm.dirty_ratio

定义以脏内存百分比为单位的字节/阈值，在该阈值中，生成脏数据的进程会在 **write()** 函数中启动。

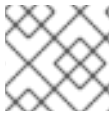


注意

一个 **vm.dirty_bytes** 或 **vm.dirty_ratio** 可以在同一时间被指定。

vm.dirty_background_bytes / vm.dirty_background_ratio

定义以脏内存百分比为单位的字节/阈值，达到此阈值时内核会尝试主动将脏数据写入硬盘。



注意

一个 **vm.dirty_background_bytes** 或 **vm.dirty_background_ratio** 可以一次指定。

vm.dirty_writeback_centisecs

定义负责将脏数据写入硬盘的内核线程定期唤醒之间的时间间隔。
这个内核参数以 100 分之一秒为单位。

vm.dirty_expire_centisecs

定义脏数据足够旧的时间以便写入硬盘。
这个内核参数以 100 分之一秒为单位。

其它资源

- [dirty pagecache writeback 和 vm.dirty 参数](#)

第 13 章 内核日志记录入门

日志文件是包含有关系统的消息的文件，包括内核、服务及其上运行的应用。Red Hat Enterprise Linux 中的日志记录系统基于内置的 **syslog** 协议。各种实用程序使用此系统记录事件并将其整理到日志文件中。这些文件在审核操作系统或故障排除问题时非常有用。

13.1. 什么是内核环缓冲

在引导过程中，控制台提供有关系统启动初始阶段的许多重要信息。为避免丢失早期消息，内核会利用称为环缓冲的早期消息。此缓冲区会保存由内核代码中的 **printk()** 函数所产生的所有消息（包括引导消息）。来自内核环缓冲的消息随后由 **syslog** 服务读取并存储在永久存储上的日志文件中。

上面提到的缓冲区是具有固定大小的循环数据结构，并且硬编码到内核中。用户可以通过 **dmesg** 命令或 **/var/log/boot.log** 文件显示存储在环缓冲中的数据。当环形缓冲区满时，新数据将覆盖旧数据。

其它资源

- **syslog(2)** 和 **dmesg(1)** 手册页

13.2. 日志级别和内核日志记录上的 PRINTK 角色

内核报告的每条消息都有一个与它关联的日志级别，用于定义消息的重要性。如[什么是内核环缓冲区](#)中所述，内核环缓冲会收集所有日志级别的内核消息。**kernel.printk** 参数用于定义缓冲区中哪些消息打印到控制台中。

日志级别值按以下顺序划分：

0

内核紧急情况。系统无法使用。

1

内核警报。必须立即采取行动。

2

内核情况被视为是关键的。

3

常见内核错误情况。

4

常见内核警告情况。

5

内核注意到一个正常但严重的情况。

6

内核信息性消息。

7

内核调试级别消息。

默认情况下，RHEL 8 中的 **kernel.printk** 包含以下四个值：

```
# sysctl kernel.printk
kernel.printk = 7 4 1 7
```

四个值按顺序定义了以下情况：

1. 控制台日志级别，定义打印到控制台的消息的最低优先级。
2. 消息没有显式附加日志级别的默认日志级别。
3. 为控制台日志级别设置最低的日志级别配置。
4. 在引导时为控制台日志级别设置默认值。

以上每个值都定义了处理错误消息的不同规则。



重要

默认的 **7 4 1 7 printk** 值可以更好地调试内核活动。但是，当与串行控制台耦合时，这个 **printk** 设置可能会导致大量 I/O 突发，这可能会导致 RHEL 系统变得暂时不响应。为避免这种情况，把 **printk** 值设置为 **4 4 1 7** 通常可以正常工作，但其代价是丢失额外的调试信息。

另请注意，某些内核命令行参数（如 **quiet** 或 **debug**）会更改默认的 **kernel.printk** 值。

其它资源

- [syslog\(2\) 手册页](#)

第 14 章 安装 KDUMP

在 RHEL 8 安装的新版本上，默认安装并激活 **kdump** 服务。使用提供的信息和流程，了解 **kdump** 是什么，以及如何 **kdump** 默认没启用，如何安装它。

14.1. 什么是 KDUMP

kdump 是一个提供崩溃转储机制，并生成一个转储文件的服务，称为崩溃转储或 **vmcore** 文件。**vmcore** 文件包含系统内存的内容，有助于分析和故障排除。**kdump** 使用 **kexec** 系统调用引导到第二个内核，这是一个不需要重启的 **捕获内核**，然后捕获崩溃内核内存的内容，并将其保存到一个文件中。第二个内核位于系统内存的保留部分。



重要

当系统出现故障时，内核崩溃转储是唯一可用的信息。因此，在关键任务环境中操作 **kdump** 非常重要。红帽建议在常规内核更新周期中定期更新和测试 **kexec-tools**。这在安装新内核功能时尤为重要。

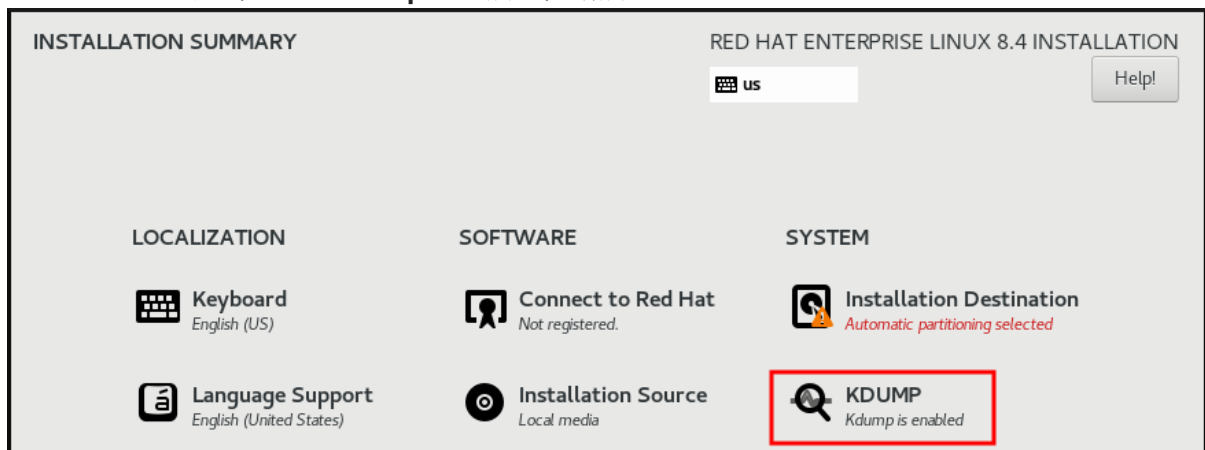
您可以为机器上所有安装的内核或者只为指定的内核启用 **kdump**。当在机器上使用多个内核时，这非常有用，有些内核足够稳定，不必担心它们会崩溃。安装 **kdump** 时，会创建一个默认的 **/etc/kdump.conf** 文件。**/etc/kdump.conf** 文件包含默认的最小 **kdump** 配置，您可以编辑该文件来自定义 **kdump** 配置。

14.2. 使用 ANACONDA 安装 KDUMP

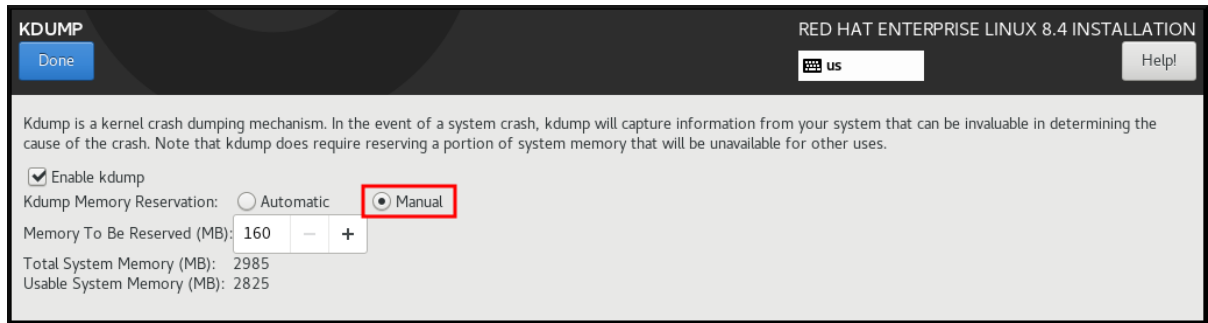
Anaconda 安装程序在交互安装过程中为 **kdump** 配置提供了一个图形界面屏幕。安装程序屏幕标题为 **KDUMP**，在主 **Installation Summary** 屏幕中提供。您可以启用 **kdump**，并保留所需的内存量。

流程

1. 在 **KDUMP** 字段中，如果 **kdump** 尚未启用，请启用它。



2. 在 **Kdump Memory Reservation** 下，如果必须自定义内存保留，请选择 **Manual**'。
3. 在 **KDUMP** 字段中，在 **Memory To Beserved (MB)** 中，为 **kdump** 设置所需的内存保留。



14.3. 在命令行上安装 KDUMP

有些安装选项，比如自定义的 **Kickstart** 安装，在某些情况下，默认 **不** 安装或启用 **kdump**。如果是这种情况，请按照以下流程操作。

先决条件

- 一个有效的 RHEL 订阅。
- 包含用于您系统 CPU 架构的 **kexec-tools** 软件包的存储库。
- 满足 **kdump** 配置和目标的要求。详情请查看 [支持的 kdump 配置和目标](#)。

流程

1. 检查是否在系统上安装了 **kdump**：

```
# rpm -q kexec-tools
```

如果安装了该软件包，输出：

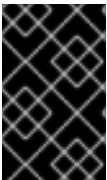
```
kexec-tools-2.0.17-11.el8.x86_64
```

如果没有安装该软件包，输出：

```
package kexec-tools is not installed
```

2. 通过以下方法安装 **kdump** 和其他必要的软件包：

```
# dnf install kexec-tools
```



重要

从 **kernel-3.10.0-693.el7** 开始，**kdump** 支持 **Intel IOMMU** 驱动程序。对于 **kernel-3.10.0-514[.XYZ].el7** 及早期版本，您必须确保 **Intel IOMMU** 被禁用，以防止无响应的捕获内核。

第 15 章 在命令行中配置 KDUMP

在系统引导过程中为 **kdump** 保留内存。内存大小是在系统的 Grand Unified Bootloader (GRUB) 配置文件中配置的。内存大小取决于配置文件中指定的 **crashkernel=** 值以及系统物理内存的大小。

15.1. 估算 KDUMP 大小

在计划和构建 **kdump** 环境时，务必要知道崩溃转储文件需要多大的空间。

makedumpfile --mem-usage 命令估计崩溃转储文件需要的空间。它生成一个内存使用率报告。这个报告帮助您确定转储级别，以及可以安全排除哪些页面。

流程

- 执行以下命令以生成一个内存使用率报告：

```
# makedumpfile --mem-usage /proc/kcore
```

TYPE	PAGES	EXCLUDABLE	DESCRIPTION
ZERO	501635	yes	Pages filled with zero
CACHE	51657	yes	Cache pages
CACHE_PRIVATE	5442	yes	Cache pages + private
USER	16301	yes	User process pages
FREE	77738211	yes	Free pages
KERN_DATA	1333192	no	Dumpable kernel data

重要

makedumpfile --mem-usage 命令会以页为单位报告所需的内存。这意味着您必须根据内核页面大小计算所使用的内存大小。

默认情况下，RHEL 内核在 AMD64 和 Intel 64 CPU 构架上使用 4 KB 大小的页，在 IBM POWER 构架上使用 64 KB 大小的页。

15.2. 配置 KDUMP 内存用量

为 **kdump** 的内存保留在系统引导过程中发生。内存大小是在系统的 Grand Unified Bootloader (GRUB) 配置中设定的。内存大小取决于配置文件中指定的 **crashkernel=** 选项的值以及系统物理内存的大小。

您可以使用多种方式定义 **crashkernel=** 选项。您可以指定 **crashkernel=** 值或配置 **auto** 选项。**crashkernel=auto** 参数根据系统中的物理内存总量自动保留内存。配置后，内核将自动为捕获内核保留适当数量的所需内存。这有助于防止内存不足(OOM)错误。

注意

kdump 的自动内存分配因系统硬件架构和可用内存大小而异。

例如，在 AMD64 和 Intel 64 上，**crashkernel=auto** 参数仅在可用内存超过 1GB 时才起作用。64 位 ARM 架构和 IBM Power 系统需要超过 2 GB 的可用内存。

如果系统自动分配低于最小内存阈值，您可以手动配置保留内存量。

先决条件

- 您在系统上具有 root 权限。
- 满足 **kdump** 配置和目标的要求。详情请查看 [支持的 kdump 配置和目标](#)。

流程

1. 准备 **crashkernel=** 选项。

- 例如：要保留 128 MB 内存，请使用：

```
crashkernel=128M
```

- 或者，您可以根据安装的内存总量将保留内存量设置为变量。变量中的内存保留语法为 **crashkernel=<range1>:<size1>,<range2>:<size2>**。例如：

```
crashkernel=512M-2G:64M,2G-:128M
```

如果系统内存总量为 512 MB 和 2 GB，则命令保留 64 MB 内存。如果内存量超过 2 GB，则内存保留为 128 MB。

- 保留内存的偏移。
有些系统需要保留具有特定固定偏移量的内存，因为 **crashkernel** 保留在早期发生，您可能需要为特殊用途保留更多内存。当您定义了一个偏移时，保留的内存将从这个偏移位置开始。要偏移保留的内存，请使用以下语法：

```
crashkernel=128M@16M
```

在这个示例中，**kdump** 从 16 MB 开始保留 128 MB 内存（物理地址 **0x01000000**）。如果将偏移参数设置为 0 或完全省略，**kdump** 会自动偏移保留内存。在设置变量内存保留时，也可以使用此语法。在这种情况下，偏移总是指定最后一个。例如：

```
crashkernel=512M-2G:64M,2G-:128M@16M
```

2. 将 **crashkernel=** 选项应用到引导装载程序配置：

```
# grubby --update-kernel=ALL --args="crashkernel=<value>"
```

将 **<value>** 替换为您在上一步中准备的 **crashkernel=** 选项的值。

其它资源

- [kdump 的内存要求](#)
- [配置内核命令行参数](#)
- [如何在系统引导前手动修改 GRUB 中的引导参数](#)
- [如何在 Red Hat Enterprise Linux 8 中安装并引导自定义内核](#)
- [grubby\(8\) 手册页](#)

15.3. 配置 KDUMP 目标

崩溃转储通常以一个文件形式存储在本地文件系统中，直接写入设备。或者，您可以为崩溃转储进行设置，以通过使用 **NFS** 或 **SSH** 协议的网络进行发送。一次只能设置其中一个选项来保留崩溃转储文件。默认行为是将其存储在本地文件系统的 `/var/crash/` 目录中。

先决条件

- 您在系统上具有 root 权限。
- 满足 **kdump** 配置和目标的要求。详情请查看 [支持的 kdump 配置和目标](#)。

流程

- 要将崩溃转储文件保存在本地文件系统的 `/var/crash/` 目录中，请编辑 `/etc/kdump.conf` 文件并指定路径：

```
path /var/crash
```

选项 `path /var/crash` 代表 **kdump** 在其中保存崩溃转储文件的文件系统的路径。



注意

- 当您在 `/etc/kdump.conf` 文件中指定转储目标时，路径是相对于指定的转储目标。
- 当您没有在 `/etc/kdump.conf` 文件中指定转储目标时，该路径表示根目录的绝对路径。

根据当前系统中挂载的内容，会自动执行转储目标和调整的转储路径。

要保护 **kdump** 生成的崩溃转储文件和附带文件，您应该为目标目的地目录设置正确的属性，如用户权限和 SELinux 上下文。另外，您可以定义一个脚本，如 `kdump.conf` 文件中的 `kdump_post.sh`，如下所示：

```
kdump_post <path_to_kdump_post.sh>
```

`kdump_post` 指令指定一个 shell 脚本或在 **kdump** 完成捕获后执行的命令，并将崩溃转储保存到指定的目的地。您可以使用此机制来扩展 **kdump** 的功能，以执行包括文件权限的调整等操作。

例 15.1. kdump 目标配置

```
# grep -v ^# /etc/kdump.conf | grep -v ^$
ext4 /dev/mapper/vg00-varcrashvol
path /var/crash
core_collector makedumpfile -c --message-level 1 -d 31
```

此处，转储目标被指定 (`ext4 /dev/mapper/vg00-varcrashvol`)，因此在 `/var/crash` 处挂载。`path` 选项也被设置为 `/var/crash`，因此 **kdump** 会将 `vmcore` 文件保存在 `/var/crash/var/crash` 目录中。

- 要更改要保存崩溃转储的本地目录，请以 **root** 用户身份编辑 `/etc/kdump.conf` 配置文件：
 - a. 从 `#path /var/crash` 行的开头删除哈希符号 (`#`)。

- b. 使用预期的目录路径替换该值。例如：

```
path /usr/local/cores
```



重要

在 RHEL 8 中，当 **kdump systemd** 服务启动时，使用 **path** 指令定义为 **kdump** 目标的目录必须存在，以避免失败。这个行为与 RHEL 之前的版本不同，其中如果服务启动时目录不存在，则会自动创建该目录。

- 要将文件写入不同的分区，请编辑 **/etc/kdump.conf** 配置文件：
 - a. 根据您的选择，从 **#ext4** 行的开头删除哈希符号(**#**)。
 - 设备名称 (**#ext4 /dev/vg/lv_kdump** 行)
 - 文件系统标签 (**#ext4 LABEL=/boot** 行)
 - UUID (**#ext4 UUID=03138356-5e61-4ab3-b58e-27507ac41937** 行)
 - b. 将文件系统类型和设备名称、标签或 UUID 改为所需的值。指定 UUID 值的正确语法是 **UUID="correct-uuid"** 和 **UUID=correct-uuid**。例如：

```
ext4 UUID=03138356-5e61-4ab3-b58e-27507ac41937
```



重要

建议您使用 **LABEL=** 或 **UUID=** 指定存储设备。无法保证 **/dev/sda3** 等磁盘设备名称在重启后保持一致。

当您在 IBM Z 硬件上使用 Direct Access Storage Device (DASD) 时，请确保在使用 **kdump** 前，转储设备已在 **/etc/dasd.conf** 中被正确指定。

- 要将崩溃转储直接写入设备，请编辑 **/etc/kdump.conf** 配置文件：
 - a. 从 **#raw /dev/vg/lv_kdump** 行的开头删除哈希符号(**#**)。
 - b. 使用预期的设备名称替换该值。例如：

```
raw /dev/sdb1
```

- 要使用 **NFS** 协议将崩溃转储保存到远程机器上：
 - a. 从 **#nfs my.server.com:/export/tmp** 行的开头删除哈希符号(**#**)。
 - b. 使用有效的主机名和目录路径替换该值。例如：

```
nfs penguin.example.com:/export/cores
```

- c. 重启 **kdump** 服务以使更改生效：

```
sudo systemctl restart kdump.service
```



注意

当使用 NFS 指令指定 NFS 目标时，**kdump.service** 会自动尝试挂载 NFS 目标来检查磁盘空间。不需要事先挂载 NFS 目标。要防止 **kdump.service** 挂载目标，请在 **kdump.conf** 中使用 **dracut_args --mount** 指令，以便 **kdump.service** 使用指定 NFS 目标的 **--mount** 参数来调用 **dracut** 工具。

- 要使用 SSH 协议将崩溃转储保存到远程机器上：
 - a. 从 **#ssh user@my.server.com** 行的开头删除哈希符号(#)。
 - b. 使用有效的用户名和密码替换该值。
 - c. 在配置中包含您的 SSH 密钥。
 - i. 从 **#sshkey /root/.ssh/kdump_id_rsa** 行的开头删除哈希符号。
 - ii. 将该值改为您要转储的服务器中有效密钥的位置。例如：

```
ssh john@penguin.example.com
sshkey /root/.ssh/mykey
```

其它资源

[第 15.8 节 “系统崩溃后 kdump 产生的文件”](#)

15.4. 配置 KDUMP 核心收集器

kdump 服务使用 **core_collector** 程序捕获崩溃转储镜像。在 RHEL 中，**makedumpfile** 工具是默认的内核收集器。它通过以下方式帮助缩小转储文件：

- 压缩崩溃转储文件的大小，并使用不同的转储级别只复制所需的页。
- 排除不必要的崩溃转储页。
- 过滤要包含在崩溃转储中的页类型。

语法

```
core_collector makedumpfile -l --message-level 1 -d 31
```

选项

- **-c**、**-l** 或 **-p**：指定每个页的压缩 dump 文件的格式，使用 **zlib** 用于 **-c** 选项、使用 **lzo** 用于 **-l** 新选项，或 **snappy** 用于 **-p** 选项。
- **-d (dump_level)**：排除页面，它们不会复制到转储文件中。
- **--message-level**：指定消息类型。您可以通过使用这个选项指定 **message_level** 来限制打印的输出。例如，把 **message_level** 设置为 7 可打印常见消息和错误消息。**message_level** 的最大值为 31

先决条件

- 您在系统上具有 root 权限。
- 满足 **kdump** 配置和目标的要求。详情请查看 [支持的 kdump 配置和目标](#)。

流程

1. 以 **root** 用户身份，编辑 `/etc/kdump.conf` 配置文件并从 `#core_collector makedumpfile -l --message-level 1 -d 31` 的开头删除 hash 符号("#")。
2. 要启用崩溃转储文件压缩，请执行：

```
core_collector makedumpfile -l --message-level 1 -d 31
```

-l 选项指定转储压缩的文件格式。**-d** 选项将转储级别指定为 31。**message-level** 选项将消息级别指定为 1。

另外，请考虑使用 **-c** 和 **-p** 选项的示例：

- 要使用 **-c** 压缩崩溃转储文件：

```
core_collector makedumpfile -c -d 31 --message-level 1
```

- 要使用 **-p** 压缩崩溃转储文件：

```
core_collector makedumpfile -p -d 31 --message-level 1
```

其它资源

- [makedumpfile\(8\) man page](#)
- [kdump 的配置文件](#)

15.5. 配置 KDUMP 默认失败响应

默认情况下，当 **kdump** 不能在配置的目标位置创建崩溃转储文件时，系统会重启，转储在此过程中会丢失。您可以更改默认故障响应，并配置 **kdump** 以执行不同的操作，以防无法将内核转储保存到主目标。额外的操作是：

dump_to_rootfs

将内核转储保存到 **root** 文件系统。

reboot

重启系统，在此过程中会丢失内核转储。

halt

停止系统，在此过程中会丢失内核转储。

poweroff

关闭系统，在此过程中会丢失内核转储。

shell

从 **initramfs** 中运行 shell 会话，您可以手动记录内核转储。

final_action

在 **kdump** 成功或在 shell 或 **dump_to_rootfs** 失败操作完成后，启用额外的操作，如 **reboot**、**halt** 和 **poweroff**。默认为 **reboot**。

failure_action

指定在内核崩溃中转储可能失败时要执行的操作。默认为 **reboot**。

先决条件

- root 权限。
- 满足 **kdump** 配置和目标的要求。详情请查看 [支持的 kdump 配置和目标](#)。

流程

1. 以 **root** 用户身份，从 `/etc/kdump.conf` 配置文件中 **#failure_action** 行的开头删除哈希符号(**#**)。
2. 将值替换为所需操作。

```
failure_action poweroff
```

其它资源

- [配置 kdump 目标](#)

15.6. KDUMP 的配置文件

kdump 内核的配置文件是 `/etc/sysconfig/kdump`。此文件控制 **kdump** 内核命令行参数。对于大多数配置，请使用默认选项。然而，在某些情况下，您可能需要修改某些参数来控制 **kdump** 内核行为。例如：修改 **KDUMP_COMMANDLINE_APPEND** 选项，以附加 **kdump** 内核命令行来获取详细的调试输出或修改 **KDUMP_COMMANDLINE_REMOVE** 选项，以从 **kdump** 命令行中删除参数。

KDUMP_COMMANDLINE_REMOVE

这个选项从当前 **kdump** 命令行中删除参数。它删除可能导致 **kdump** 错误或 **kdump** 内核引导失败的参数。这些参数可以从之前的 **KDUMP_COMMANDLINE** 进程解析，或者继承自 `/proc/cmdline` 文件。

如果未配置此变量，它将继承 `/proc/cmdline` 文件中的所有值。配置此选项还提供了有助于调试问题的信息。

要删除某些参数，请将其添加到 **KDUMP_COMMANDLINE_REMOVE** 中，如下所示：

```
KDUMP_COMMANDLINE_REMOVE="hugepages hugepagesz slub_debug quiet log_buf_len swiotlb"
```

KDUMP_COMMANDLINE_APPEND

此选项将参数附加到当前命令行。这些参数可能已被前面的 **KDUMP_COMMANDLINE_REMOVE** 变量解析。

对于 **kdump** 内核，禁用某些模块，如 **mce**、**cgroup**、**numa**，**hest_disable** 有助于防止内核错误。这些模块可能会消耗为 **kdump** 保留的大量内核内存，或导致 **kdump** 内核引导失败。

要在 **kdump** 内核命令行中禁用内存 **cgroups**，请运行以下命令：

```
KDUMP_COMMANDLINE_APPEND="cgroup_disable=memory"
```

其它资源

- `Documentation/admin-guide/kernel-parameters.txt` 文件
- `/etc/sysconfig/kdump` 文件

15.7. 测试 KDUMP 配置

配置 `kdump` 后，您必须手动测试系统崩溃，并确保在定义的 `kdump` 目标中生成了 `vmcore` 文件。`vmcore` 文件是从新引导的内核上下文中捕获的，因此具有帮助调试内核崩溃的重要信息。



警告

不要对活动状态的生产环境系统测试 `kdump`。测试 `kdump` 的命令将导致内核崩溃，且数据丢失。根据您的系统架构，确保您安排了充足的维护时间，因为 `kdump` 测试可能需要多次重启，且启动时间很长。

如果 `vmcore` 文件在 `kdump` 测试过程中没有生成，请在再次运行测试前识别并修复问题，以使 `kdump` 测试成功。

重要

确保安排了充足的维护时间，因为 `kdump` 测试可能需要多次重启，且启动时间很长。

如果进行任何手动系统修改，您必须在任何系统修改的最后测试 `kdump` 配置。例如，如果您进行任何以下更改，请确保测试 `kdump` 配置，以获得最佳 `kdump` 性能：

- 软件包升级。
- 硬件级别的更改，如存储或网络更改。
- 固件和 BIOS 升级。
- 包括第三方模块的新安装和应用程序升级。
- 如果您使用热插拔机制在支持此机制的硬件上添加更多内存。
- 在 `/etc/kdump.conf` 或 `/etc/sysconfig/kdump` 文件中进行更改后。

先决条件

- 您在系统上具有 `root` 权限。
- 您已保存了所有重要数据。测试 `kdump` 的命令导致内核崩溃及数据丢失。
- 您已根据系统架构安排了大量机器维护时间。

流程

1. 启用 `kdump` 服务：

```
# kdumpctl restart
```

2. 检查 **kdump** 服务的状态。使用 **kdumpctl** 命令，您可以在控制台上打印输出。

```
# kdumpctl status
kdump:Kdump is operational
```

或者，如果您使用 **systemctl** 命令，输出在 **systemd** 日志中打印。

3. 启动内核崩溃来测试 **kdump** 配置。**sysrq-trigger** 组合键导致内核崩溃，并在需要时可能重启系统。

```
# echo c > /proc/sysrq-trigger
```

在内核重启时，**address-YYYY-MM-DD-HH:MM:SS/vmcore** 文件在您在 **/etc/kdump.conf** 文件中指定的位置创建。默认值为 **/var/crash/**。

其它资源

- [配置 kdump 目标](#)

15.8. 系统崩溃后 KDUMP 产生的文件

系统崩溃后，**kdump** 服务在转储文件(**vmcore**)中捕获内核内存，它还生成额外的诊断文件，以帮助故障排除和事后分析。

kdump 生成的文件：

- **vmcore** - 包含崩溃时系统内存的主内核内存转储文件。它包含根据 **kdump** 配置中指定的 **core_collector** 程序配置的数据。默认情况下，内核数据结构、处理信息、堆栈跟踪和其他诊断信息。
- **vmcore-dmesg.txt** - panic 的主内核中的内核环缓冲区日志的内容(**dmesg**)。
- **kexec-dmesg.log** - 包含收集 **vmcore** 数据的二级 **kexec** 内核执行中的内核和系统日志消息。

其它资源

- [什么是内核环缓冲](#)
- [kdump](#)

15.9. 启用和禁用 KDUMP 服务

您可以进行配置，以在特定内核或所有安装的内核上启用或禁用 **kdump** 功能。您必须定期测试 **kdump** 功能，并验证它是否工作正常。

先决条件

- 您在系统上具有 root 权限。
- 您已为配置和目标完成了 **kdump** 要求。请参阅 [支持的 kdump 配置和目标](#)。
- 用于安装 **kdump** 的所有配置都是根据需要设置的。

流程

- 为 **multi-user.target** 启用 **kdump** 服务：

```
# systemctl enable kdump.service
```

- 在当前会话中启动服务：

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

- 停止 **kdump** 服务：

```
# systemctl stop kdump.service
```

- 禁用 **kdump** 服务：

```
# systemctl disable kdump.service
```



警告

建议将 **kptr_restrict=1** 设置为默认值。当将 **kptr_restrict** 设置为 (1) 作为默认时，**kdumpctl** 服务会加载崩溃内核，即使启用了内核地址空间布局 (KASLR)。

如果 **kptr_restrict** 没有被设置为 1，并且 KASLR 被启用了，则生成的 **/proc/kcore** 文件的内容全为零。**kdumpctl** 服务无法访问 **/proc/kcore** 文件，并加载崩溃内核。**kexec-kdump-howto.txt** 文件显示一条警告信息，建议您设置 **kptr_restrict=1**。验证 **sysctl.conf** 文件中的以下内容，以确保 **kdumpctl** 服务加载崩溃内核：

- **sysctl.conf** 文件中的内核 **kptr_restrict=1**。

15.10. 防止内核驱动程序为 KDUMP 加载

您可以通过在 **/etc/sysconfig/kdump** 配置文件中添加 **KDUMP_COMMANDLINE_APPEND=** 变量来从加载某些内核驱动程序中控制捕获内核。使用这个方法，您可以防止来自加载指定的内核模块中的 **kdump** 初始 RAM 磁盘镜像 **initramfs**。这有助于防止内存不足(OOM) killer 错误或其他崩溃内核失败。

您可以使用以下配置选项之一附加 **KDUMP_COMMANDLINE_APPEND=** 变量：

- **rd.driver.blacklist=<modules>**
- **modprobe.blacklist=<modules>**

先决条件

- 您在系统上具有 root 权限。

流程

1. 显示载入到当前运行内核的模块的列表。选择您要阻止其加载的内核模块。

```
$ lsmod

Module              Size Used by
fuse                 126976 3
xt_CHECKSUM          16384 1
ipt_MASQUERADE       16384 1
uinput              20480 1
xt_contrack          16384 1
```

2. 更新 `/etc/sysconfig/kdump` 文件中的 `KDUMP_COMMANDLINE_APPEND=` 变量。例如：

```
KDUMP_COMMANDLINE_APPEND="rd.driver.blacklist=hv_vmbus,hv_storvsc,hv_utils,hv_net
vsc,hid-hyperv"
```

另外，考虑以下使用 `modprobe.blacklist= <modules>` 配置选项的示例：

```
KDUMP_COMMANDLINE_APPEND="modprobe.blacklist=emcp modprobe.blacklist=bnx2fc
modprobe.blacklist=libfcoe modprobe.blacklist=fcoe"
```

3. 重启 `kdump` 服务：

```
# systemctl restart kdump
```

其它资源

- [dracut.cmdline](#) 手册页

15.11. 在使用加密磁盘的系统中运行 KDUMP

当您运行 LUKS 加密的分区时，系统需要一定数量的可用内存。如果系统可用内存量小于所需的可用内存量，则 `cryptsetup` 实用程序无法挂载分区。因此，在第二个内核（捕获内核）中将 `vmcore` 文件捕获到加密的目标位置会失败。

`kdumpctl estimate` 命令可帮助您估算 `kdump` 所需的内存量。`kdumpctl estimate` 打印推荐的 `crashkernel` 值，这是 `kdump` 所需的最合适的内存大小。

推荐的 `crashkernel` 值是根据当前的内核大小、内核模块、`initramfs` 以及 LUKS 加密的目标内存要求计算的。

如果您使用自定义 `crashkernel=` 选项，`kdumpctl estimate` 会打印 `LUKS required size` 值。值是 LUKS 加密目标所需的内存大小。

流程

1. 输出估计的 `crashkernel=` 值：

```
# *kdumpctl estimate*

Encrypted kdump target requires extra memory, assuming using the keyslot with minimum
memory requirement
Reserved crashkernel: 256M
```

Recommended crashkernel: 652M

Kernel image size: 47M

Kernel modules size: 8M

Initramfs size: 20M

Runtime reservation: 64M

LUKS required size: 512M

Large modules: <none>

WARNING: Current crashkernel size is lower than recommended size 652M.

2. 通过增加 **crashkernel=** 值来配置所需的内存量。
3. 重启系统。



注意

如果 **kdump** 服务仍无法将转储文件保存到加密的目标，请根据需要增大 **crashkernel=** 值。

第 16 章 启用 KDUMP

对于 RHEL 8 系统，您可以配置在特定内核或所有安装的内核上启用或禁用 **kdump** 功能。但是，您必须定期测试 **kdump** 功能，并验证它是否工作正常。

16.1. 为所有安装的内核启用 KDUMP

在 **kexec** 工具安装后，**kdump** 服务通过启用 **kdump.service** 启动。您可以为在机器上安装的所有内核启用并启动 **kdump** 服务。

先决条件

- 有管理员特权。

流程

1. 将 **crashkernel=** 命令行参数添加到所有安装的内核中。

```
# grubby --update-kernel=ALL --args="crashkernel=xxM"
```

xxM 是所需的内存（以 MB 为单位）。

2. 启用 **kdump** 服务。

```
# systemctl enable --now kdump.service
```

验证

- 检查 **kdump** 服务是否正在运行。

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

```
○ kdump.service - Crash recovery kernel arming
```

```
Loaded: loaded (/usr/lib/systemd/system/kdump.service; enabled; vendor preset:  
disabled)
```

```
Active: active (live)
```

16.2. 为特定安装的内核启用 KDUMP

您可以为机器上的特定内核启用 **kdump** 服务。

先决条件

- 有管理员特权。

流程

1. 列出安装在机器上的内核。

```
# ls -a /boot/vmlinuz-  
/boot/vmlinuz-0-rescue-2930657cd0dc43c2b75db480e5e5b4a9  
/boot/vmlinuz-4.18.0-330.el8.x86_64
```

```
/boot/vmlinuz-4.18.0-330.rt7.111.el8.x86_64
```

- 向系统的 Grand Unified Bootloader (GRUB)配置中添加特定的 **kdump** 内核。
例如：

```
# grubby --update-kernel=vmlinuz-4.18.0-330.el8.x86_64 --args="crashkernel=xxM"
```

xxM 是所需的内存保留（以 MB 为单位）。

- 启用 **kdump** 服务。

```
# systemctl enable --now kdump.service
```

验证

- 检查 **kdump** 服务是否正在运行。

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

```
○ kdump.service - Crash recovery kernel arming
```

```
Loaded: loaded (/usr/lib/systemd/system/kdump.service; enabled; vendor preset:  
disabled)
```

```
Active: active (live)
```

16.3. 禁用 KDUMP 服务

您可以停止 **kdump.service**，并在 RHEL 8 系统上禁用该服务。

先决条件

- 满足 **kdump** 配置和目标的要求。详情请查看 [支持的 kdump 配置和目标](#)。
- 安装 **kdump** 的所有配置都是根据您的需要设置的。详情请参阅 [安装 kdump](#)。

流程

- 要在当前会话中停止 **kdump** 服务：

```
# systemctl stop kdump.service
```

- 要禁用 **kdump** 服务：

```
# systemctl disable kdump.service
```



警告

建议将 `kptr_restrict=1` 设置为默认值。当 `kptr_restrict` 被默认设置为(1)时，`kdumpctl` 服务加载崩溃内核，即使 Kernel Address Space Layout (**KASLR**)被启用了或没有启用。

如果 `kptr_restrict` 没有设置为 **1**，且 **KASLR** 被启用了，则 `/proc/kcore` 文件的内容被生成为全零。`kdumpctl` 服务无法访问 `/proc/kcore` 文件，并加载崩溃内核。`kexec-kdump-howto.txt` 文件显示一条警告信息，建议您设置 `kptr_restrict=1`。验证 `sysctl.conf` 文件中的以下内容，以确保 `kdumpctl` 服务加载崩溃内核：

- `sysctl.conf` 文件中的内核 `kptr_restrict=1`。

其它资源

- [管理 systemd](#)

第 17 章 在 WEB 控制台中配置 KDUMP

您可以使用 RHEL 8 web 控制台设置并测试 **kdump** 配置。Web 控制台可以在引导时启用 **kdump** 服务。另外，web 控制台允许您为 **kdump** 配置保留内存，并以未压缩或压缩格式选择 **vmcore** 保存位置。

17.1. 在 WEB 控制台中配置 KDUMP 内存用量和目标位置

您可以为 **kdump** 内核配置内存保留，并指定目标位置，来使用 RHEL web 控制台界面捕获 **vmcore** 转储文件。

先决条件

- 必须安装并可以访问 Web 控制台。
详情请参阅[安装 Web 控制台](#)。

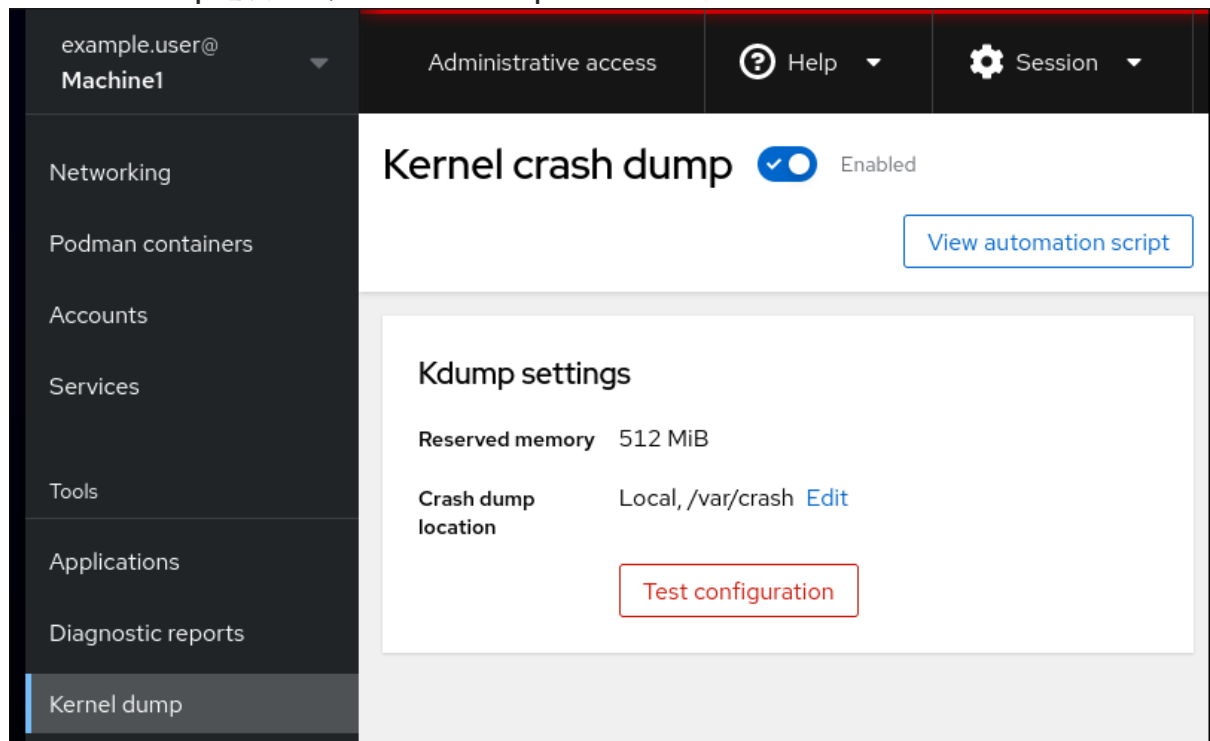
流程

1. 在 web 控制台中，打开 **Kernel dump** 选项卡，并通过将 **Kernel crash dump** 开关设置为 on 来启动 **kdump** 服务。
2. 在终端中配置 **kdump** 内存用量，例如：

```
$ sudo grubby --update-kernel ALL --args crashkernel=512M
```

重启系统以应用更改。

3. 在 **Kernel dump** 选项卡中，点 **Crash dump location** 字段末尾的 **Edit**。



4. 指定保存 **vmcore** 转储文件的目标目录：
 - 对于本地文件系统，从下拉菜单中选择 **Local Filesystem**。

Crash dump location

Location Local filesystem ▼

Directory /var/crash

Compression Compress crash dumps to save space

Apply Cancel

- 对于使用 SSH 协议的远程系统，从下拉菜单中选择 **Remote over SSH**，并指定以下字段：
 - 在 **Server** 字段中，输入远程服务器地址。
 - 在 **SSH key** 字段中，输入 SSH 密钥位置。
 - 在 **Directory** 字段中，输入目标目录。
- 对于使用 NFS 协议的远程系统，从下拉菜单中选择 **Remote over NFS**，并指定以下字段：
 - 在 **Server** 字段中，输入远程服务器地址。
 - 在 **Export** 字段中，输入 NFS 服务器的共享文件夹的位置。
 - 在 **Directory** 字段中，输入目标目录。



注意

您可以选择 **Compression** 复选框来减小 **vmcore** 文件的大小。

5. 可选：点 **View automation script** 显示自动化脚本。
此时会打开一个带有生成的脚本的窗口。您可以在 shell 脚本页和 Ansible playbook 生成选项页之间转换。
6. 可选：点 **Copy to clipboard** 复制脚本。
您可以使用此脚本在多台机器上应用相同的配置。

验证

1. 单击 **Test configuration**。

Kdump settings

Reserved memory 512 MiB

Crash dump location Local, /var/crash [Edit](#)

[Test configuration](#)

2. 在 Test kdump settings 下点 **Crash system**。



警告

当您启动系统崩溃时，内核操作会停止，并导致系统因数据丢失而崩溃。

其它资源

- [支持的 kdump 目标](#)

第 18 章 支持的 KDUMP 配置和目标

kdump 机制是 Linux 内核的一个功能，它在发生内核崩溃时生成一个崩溃转储文件。内核转储文件有关键的信息，可帮助分析和确定内核崩溃的根本原因。崩溃可能是因为各种因素，举几个例子，如硬件问题或第三方内核模块问题。

使用提供的信息和流程，您可以了解 RHEL 8 系统上支持的配置和目标，正确配置 **kdump**，并验证它是否工作正常。

18.1. KDUMP 的内存要求

要让 **kdump** 捕获内核崩溃转储，并保存它以便进一步分析，应该为捕获内核永久保留系统内存的一部分。保留时，主内核无法使用系统内存的这一部分。

内存要求因某些系统参数而异。主要因素之一就是系统的硬件构架。要找出确切的机器架构（如 Intel 64 和 AMD64，也称为 x86_64）并将其输出到标准输出，请使用以下命令：

```
$ uname -m
```

使用上述最小内存要求的列表，您可以设置合适的内存大小，来在最新可用版本上为 **kdump** 自动保留内存。内存大小取决于系统的架构和总可用物理内存。

表 18.1. kdump 所需的最小保留内存量

构架	可用内存	最小保留内存
AMD64 和 Intel 64 (x86_64)	1 GB 到 4 GB	192 MB 内存
	4 GB 到 64 GB	256 MB 内存
	64 GB 及更多	512 MB 内存
64 位 ARM 架构 (arm64)	2 GB 及更多	480 MB 内存
IBM Power 系统 (ppc64le)	2 GB 到 4 GB	384 MB 内存
	4 GB 到 16 GB	512 MB 内存
	16 GB 到 64 GB	1 GB 内存
	64 GB 到 128 GB	2 GB 内存
	128 GB 及更多	4 GB 内存
IBM Z (s390x)	1 GB 到 4 GB	192 MB 内存
	4 GB 到 64 GB	256 MB 内存
	64 GB 及更多	512 MB 内存

在很多系统中，**kdump** 可以估算所需内存量并自动保留。默认情况下，此行为是启用的，但仅适用于内存总量超过特定数量的系统，这些内存因系统架构而异。



重要

根据系统中内存总量自动配置保留内存是最佳工作量估算。实际需要的内存可能因其它因素（如 I/O 设备）而有所不同。使用内存不足将导致 debug 内核无法在内核 panic 的情况下作为捕获内核引导。要避免这个问题，请足够增大崩溃内核内存。

其它资源

- [RHEL8 次要版本间 crashkernel 参数有什么变化？](#)
- [技术能力和限制表](#)
- [自动内存保留的最小阈值](#)

18.2. 自动内存保留的最小阈值

kexec-tools 程序默认配置 **crashkernel** 命令行参数，并为 **kdump** 保留特定内存量。但是在某些系统上，仍然可以通过在引导装载程序配置文件中 **使用 crashkernel=auto** 参数为 **kdump** 分配内存，也可以在图形配置工具中启用这个选项。要使此自动保留正常工作，系统中需要有一定数量的总内存可用。内存要求因系统架构而异。如果系统内存小于指定的阈值，则您必须手动配置内存。

表 18.2. 自动保留内存所需的最小内存量

构架	所需的内存
AMD64 和 Intel 64 (x86_64)	2 GB
IBM Power 系统 (ppc64le)	2 GB
IBM Z (s390x)	4 GB



注意

RHEL 9 及更新的版本不再支持引导命令行中的 **crashkernel=auto** 选项。

18.3. 支持的 KDUMP 目标

当发生内核崩溃时，操作系统会将转储文件保存在配置的或默认的目标位置上。您可以将转储文件直接保存到设备上，将其作为文件存储在本地文件系统中，或者通过网络发送转储文件。使用以下转储目标的列表，您可以知道 **kdump** 目前支持的或不支持的目标。

表 18.3. RHEL 8 上的 **kdump** 目标

目标类型	支持的目标	不支持的目标
------	-------	--------

目标类型	支持的目标	不支持的目标
物理存储	<ul style="list-style-type: none"> ● 逻辑卷管理器(LVM)。 ● 精简配置卷。 ● 光纤通道(FC)磁盘, 如 qla2xxx、lpfc、bnx2fc 和 bfa。 ● 网络存储服务器上 iSCSI 软件配置的逻辑设备。 ● mdraid 子系统作为软件 RAID 解决方案。 ● 硬件 RAID, 如 cciss、hpsa、megaraid_sas、mpt2sas 和 aacraid。 ● SCSI 和 SATA 磁盘。 ● iSCSI 和 HBA 卸载。 ● 硬件 FCoE, 如 qla2xxx 和 lpfc。 	<ul style="list-style-type: none"> ● BIOS RAID。 ● 带有 iBFT 的软件 iSCSI. 目前支持的传输有 bnx2i、cxgb3i 和 cxgb4i。 ● 带有混合设备驱动程序的软件 iSCSI, 如 be2iscsi。 ● 以太网上的光纤通道 (FCoE)。 ● 传统 IDE。 ● GlusterFS 服务器。 ● GFS2 文件系统。 ● 集群逻辑卷管理器 (CLVM)。 ● 高可用性 LVM 卷(HA-LVM)。
网络	<ul style="list-style-type: none"> ● 使用内核模块的硬件 : tg3、igb、ixgbe、sfc、e1000e、bna、cnic、netxen_nic、qlge、bnx2x、bnx、qlcnic、be2net、enic、virtio-net、ixgbev、igbvf。 ● IPv4 协议。 ● 绑定在不同设备上的网络, 如以太网设备或 VLAN。 ● VLAN 网络。 ● 网桥。 ● 网络 Team 。 ● 绑定上标记的 VLAN 和 VLAN。 ● 绑定、team 和 VLAN 上的桥接网络。 	<ul style="list-style-type: none"> ● IPv6 协议。 ● 无线连接。 ● InfiniBand 网络。 ● 网桥和 team 上的 VLAN 网络。

目标类型	支持的目标	不支持的目标
虚拟机监控程序 (Hypervisor)	<ul style="list-style-type: none"> ● 基于内核的虚拟机 (KVM)。 ● 仅某些配置中的 Xen hypervisor。 ● VMware ESXi 4.1 和 5.1。 ● 仅在 RHEL Gen1 UP 客户机上的 Hyper-V 2012 R2。 	
文件系统	ext[234]、XFS 和 NFS 文件系统。	Btrfs 文件系统。
固件	<ul style="list-style-type: none"> ● 基于 BIOS 的系统。 ● UEFI 安全引导。 	

其它资源

- [配置 kdump 目标](#)

18.4. 支持的 KDUMP 过滤等级

要缩小转储文件的大小，**kdump** 使用 **makedumpfile** 内核收集器压缩数据，并排除不需要的信息，例如，您可以使用 **-8** 级别来删除 **hugepages** 和 **hugetlbfs** 页。**makedumpfile** 当前支持的级别可在 *Filtering levels for `kdump`* 表中看到。

表 18.4. **kdump** 的过滤级别

选项	描述
1	零页
2	缓存页
4	缓存私有
8	用户页
16	可用页

其它资源

- [配置内核收集器](#)

18.5. 支持的默认故障响应

默认情况下，当 **kdump** 创建内核转储失败时，操作系统会重启。但是，您可以将 **kdump** 配置为在将内核转储保存到主目标时执行不同的操作。

表 18.5. **kdump** 的失败响应

选项	描述
dump_to_rootfs	尝试将内核转储保存到 root 文件系统。这个选项在与网络目标合并时特别有用：如果网络目标无法访问，这个选项配置 kdump 以在本地保存内核转储。之后会重启该系统。
reboot	重启系统，这个过程会丢失 core 转储文件。
halt	关闭系统，这个过程会丢失 core 转储文件。
poweroff	关闭系统，这个此过程会丢失 core 转储。
shell	从 initramfs 内运行 shell 会话，允许用户手动记录核心转储。
final_action	在 kdump 成功，或 shell 或 dump_to_rootfs 失败操作完成后，启用额外的操作，如 reboot 、 halt 和 poweroff 操作。默认的 final_action 选项为 reboot 。

其它资源

- [配置 **kdump** 默认失败响应](#)

18.6. 使用 FINAL_ACTION 参数

当 **kdump** 成功或者 **kdump** 无法在配置的目标处保存 **vmcore** 文件时，您可以使用 **final_action** 参数执行额外的操作，如 **reboot**、**halt** 和 **poweroff**。如果没有指定 **final_action** 参数，则 **reboot** 是默认的响应。

流程

1. 要配置 **final_action**，请编辑 **/etc/kdump.conf** 文件并添加以下选项之一：

- **final_action reboot**
- **final_action halt**
- **final_action poweroff**

2. 重启 **kdump** 服务，以使更改生效。

```
# kdumpctl restart
```

18.7. 使用 FAILURE_ACTION 参数

failure_action 参数指定在内核崩溃时转储失败时要执行的操作。**failure_action** 的默认操作是 **reboot**，这会重启系统。

参数接受以下要执行的操作：

reboot

转储失败后重启系统。

dump_to_rootfs

当配置了非 root 转储目标时，将转储文件保存在 root 文件系统中。

halt

关闭系统。

poweroff

停止系统上正在运行的操作。

shell

在 **initramfs** 中启动 shell 会话，您可以从中手动执行其他恢复操作。

流程

1. 要将操作配置为在转储失败时执行的操作，请编辑 **/etc/kdump.conf** 文件并指定其中一个 **failure_action** 选项：
 - **failure_action reboot**
 - **failure_action halt**
 - **failure_action poweroff**
 - **failure_action shell**
 - **failure_action dump_to_rootfs**
2. 重启 **kdump** 服务，以使更改生效。

```
# kdumpctl restart
```

第 19 章 固件支持的转储机制

固件支持的转储 (fadump) 是一个转储捕获机制，作为 IBM POWER 系统中 **kdump** 机制的替代选择。**kexec** 和 **kdump** 机制可用于在 AMD64 和 Intel 64 系统中捕获内核转储。但是，一些硬件（如小型系统和大型机计算机）利用板载固件隔离内存区域，并防止意外覆盖对崩溃分析很重要的数据。**fadump** 工具针对 **fadump** 机制及其在 IBM POWER 系统上与 RHEL 的集成进行了优化。

19.1. IBM POWERPC 硬件支持转储固件

fadump 实用程序从带有 PCI 和 I/O 设备的完全重设系统中捕获 **vmcore** 文件。这种机制使用固件在崩溃期间保留内存区域，然后重复使用 **kdump** 用户空间脚本保存 **vmcore** 文件。内存区域由所有系统内存内容组成，但引导内存、系统注册和硬件页面表条目 (PTE) 除外。

fadump 机制通过重新引导分区并使用新内核转储之前内核崩溃中的数据，提供比传统转储类型的更高可靠性。**fadump** 需要一个基于 IBM POWER6 处理器或更高版本的硬件平台。

有关 **fadump** 机制的详情，包括针对 PowerPC 重置硬件的方法，请查看 `/usr/share/doc/kexec-tools/fadump-howto.txt` 文件。



注意

未保留的内存区域（称为引导内存）是在崩溃事件后成功引导内核所需的 RAM 量。默认情况下，引导内存大小为 256MB 或系统 RAM 总量的 5%，以较大者为准。

与 **kexec-initiated** 事件不同，**fadump** 机制使用 production 内核恢复崩溃转储。崩溃后引导时，PowerPC 硬件使设备节点 `/proc/device-tree/rtas/ibm.kernel-dump` 可供 **proc** 文件系统 (**procf**s) 使用。**fadump-aware kdump** 脚本，检查存储的 **vmcore**，然后完全完成系统重启。

19.2. 启用固件支持的转储机制

您可以通过启用固件支持的转储(**fadump**)机制来增强 IBM POWER 系统的崩溃转储功能。

在安全引导环境中，**GRUB2** 引导装载程序分配引导内存区域，称为 Real Mode Area (RMA)。RMA 的大小为 512 MB，它在引导组件中划分；如果组件超过了其大小分配，则 **GRUB2** 会失败，并显示内存不足 (**OOM**) 错误。



警告

不要在 RHEL 8.7 和 8.6 版本的安全引导环境中启用固件支持的转储(**fadump**)机制。**GRUB2** 引导装载程序失败，并显示以下错误：

```
error: .././grub-core/kern/mm.c:376:out of memory.
Press any key to continue...
```

仅当您因为 **fadump** 配置而增加默认 **initramfs** 大小时，系统才是可恢复的。

有关恢复系统的临时解决方案方法的详情，请参考 [GRUB 内存不足\(OOM\)中的系统引导结束](#) 文章。

流程

1. 安装和配置 **kdump**。

2. 启用 **fadump=on** 内核选项：

```
# grubby --update-kernel=ALL --args="fadump=on"
```

3. (可选) 如果要指定保留引导内存而不是使用默认值, 启用 **crashkernel=xxM** 选项, 其中 **xx** 是所需的内存量 (以 MB 为单位)：

```
# grubby --update-kernel=ALL --args="crashkernel=xxM fadump=on"
```



重要

在指定引导选项时, 请在执行前测试所有引导选项。如果 **kdump** 内核无法引导, 请逐渐增加 **crashkernel=** 参数中指定的值来设置适当的值。

19.3. IBM Z 硬件支持的固件转储机制

IBM Z 系统支持以下固件支持的转储机制：

- 独立转储 (**sadump**)
- **VMDUMP**

IBM Z 系统支持并使用 **kdump** 基础架构。但是, 使用 IBM Z 的固件支持的转储 (**fadump**) 方法之一可以提供各种优点：

- **sadump** 机制是从系统控制台启动和控制的, 并存储在 **IPL** 可引导设备中。
- **VMDUMP** 机制与 **sadump** 类似。此工具也从系统控制台启动, 但会从硬件检索生成的转储并将其复制到系统以进行分析。
- 这些方法 (与其他基于硬件的转储机制类似) 能够在 **kdump** 服务启动前捕获机器在早期启动阶段的状态。
- 尽管 **VMDUMP** 包含将转储文件接收到 Red Hat Enterprise Linux 系统中的机制, 但 **VMDUMP** 的配置和控制是从 IBM Z 硬件控制台管理的。

其它资源

- [独立转储程序](#)
- [使用 VMDUMP 在 z/VM 中创建转储](#)
- [在 Red Hat Enterprise Linux 7.4 上使用转储工具](#)

19.4. 在 FUJITSU PRIMEQUEST 系统中使用 SADUMP

Fujitsu **sadump** 机制旨在为 **kdump** 无法成功完成时提供 **fallback** 转储捕获。**sadump** 机制是从系统管理板 (MMB) 接口手动调用的。使用 MMB, 为 Intel 64 或 AMD 64 服务器配置 **kdump**, 然后启用 **sadump**。

注意

1. 在 `/etc/sysctl.conf` 文件中添加或编辑以下行，以确保 **sadump** 的 **kdump** 按预期启动：

```
kernel.panic=0
kernel.unknown_nmi_panic=1
```



警告

特别是，请确保在 **kdump** 后系统不会重启。如果系统在 **kdump** 无法保存 **vmcore** 文件后重启，则无法调用 **sadump**。

2. 适当地将 `/etc/kdump.conf` 中的 `failure_action` 参数设置为 **halt** 或 **shell**。

```
failure_action shell
```

其它资源

- FUJITSU Server PRIMEQUEST 2000 系列安装手册

第 20 章 分析内核转储

要确定系统崩溃的原因，您可以使用 **crash** 实用程序，它提供了一个与 GNU Debugger (GDB) 类似的交互式提示符。这个工具允许您交互式地分析由 **kdump**、**netdump**、**diskdump** 或 **xendump** 创建的内核转储以及正在运行的 Linux 系统。另外，您还可以使用 Kernel Oops Analyzer 或者 Kdump Helper 工具。

20.1. 安装 CRASH 工具

使用提供的信息，了解所需的软件包，以及安装 **crash** 工具的流程。默认情况下，**crash** 工具可能没有在 RHEL 8 系统上安装。**crash** 是一个工具，来在系统运行时或内核崩溃发生及内核转储文件创建后，以交互方式分析系统状态。内核转储文件也称为 **vmcore** 文件。

流程

1. 启用相关的软件仓库：

```
# subscription-manager repos --enable baseos repository
```

```
# subscription-manager repos --enable appstream repository
```

```
# subscription-manager repos --enable rhel-8-for-x86_64-baseos-debug-rpms
```

2. 安装 **crash** 软件包：

```
# yum install crash
```

3. 安装 **kernel-debuginfo** 软件包：

```
# yum install kernel-debuginfo
```

软件包 **kernel-debuginfo** 将对应于正在运行的内核，并提供用于转储分析所需的数据。

20.2. 运行和退出 CRASH 工具

使用提供的信息，了解所需的参数，以及运行并退出 **crash** 工具的流程。**crash** 是一个工具，来在系统运行时或内核崩溃发生及内核转储文件创建后，以交互方式分析系统状态。内核转储文件也称为 **vmcore** 文件。

先决条件

- 确定当前运行的内核（例如 **4.18.0-5.el8.x86_64**）。

流程

1. 要启动 **crash** 工具，请传递以下两个必要的参数：

- **debug-info**（解压缩的 **vmlinuz** 镜像），如 **/usr/lib/debug/lib/modules/4.18.0-5.el8.x86_64/vmlinuz**，通过特定的 **kernel-debuginfo** 软件包提供。

- 实际 **vmcore** 文件，如 **/var/crash/127.0.0.1-2018-10-06-14:05:33/vmcore** 生成的 **crash** 命令如下所示：

```
# crash /usr/lib/debug/lib/modules/4.18.0-5.el8.x86_64/vmlinux /var/crash/127.0.0.1-2018-10-06-14:05:33/vmcore
```

使用 **kdump** 捕获的相同 *<kernel>* 版本。

例 20.1. 运行 crash 工具

以下示例演示了使用 4.18.0-5.el8.x86_64 内核分析在 2018 年 10 月 6 日下午 14:05 时创建的内核转储。

```
...
WARNING: kernel relocated [202MB]: patching 90160 gdb minimal_symbol values

    KERNEL: /usr/lib/debug/lib/modules/4.18.0-5.el8.x86_64/vmlinux
    DUMPFILE: /var/crash/127.0.0.1-2018-10-06-14:05:33/vmcore [PARTIAL DUMP]
    CPUS: 2
    DATE: Sat Oct 6 14:05:16 2018
    UPTIME: 01:03:57
    LOAD AVERAGE: 0.00, 0.00, 0.00
    TASKS: 586
    NODENAME: localhost.localdomain
    RELEASE: 4.18.0-5.el8.x86_64
    VERSION: #1 SMP Wed Aug 29 11:51:55 UTC 2018
    MACHINE: x86_64 (2904 Mhz)
    MEMORY: 2.9 GB
    PANIC: "sysrq: SysRq : Trigger a crash"
    PID: 10635
    COMMAND: "bash"
    TASK: ffff8d6c84271800 [THREAD_INFO: ffff8d6c84271800]
    CPU: 1
    STATE: TASK_RUNNING (SYSRQ)

crash>
```

- 要退出交互式提示并停止 **crash**，请输入 **exit** 或 **q**。

例 20.2. 退出 crash 工具

```
crash> exit
~]#
```



注意

crash 命令也可以用作调试实时系统的强大工具。但是请谨慎使用它，以免破坏您的系统。

其它资源

- [预期系统重启指南](#)

20.3. 在 CRASH 工具中显示各种指示符

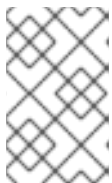
使用 **crash** 工具来显示各种指示符，如内核消息缓冲区、回溯追踪、进程状态、虚拟内存信息以及打开文件。

显示消息缓冲

- 要显示内核消息缓冲区，请在互动提示符下输入 **log** 命令：

```
crash> log
... several lines omitted ...
EIP: 0060:[<c068124f>] EFLAGS: 00010096 CPU: 2
EIP is at sysrq_handle_crash+0xf/0x20
EAX: 00000063 EBX: 00000063 ECX: c09e1c8c EDX: 00000000
ESI: c0a09ca0 EDI: 00000286 EBP: 00000000 ESP: ef4dbf24
DS: 007b ES: 007b FS: 00d8 GS: 00e0 SS: 0068
Process bash (pid: 5591, ti=ef4da000 task=f196d560 task.ti=ef4da000)
Stack:
c068146b c0960891 c0968653 00000003 00000000 00000002 efade5c0 c06814d0
<0> ffffffff c068150f b7776000 f2600c40 c0569ec4 ef4dbf9c 00000002 b7776000
<0> efade5c0 00000002 b7776000 c0569e60 c051de50 ef4dbf9c f196d560 ef4dbfb4
Call Trace:
[<c068146b>] ? __handle_sysrq+0xfb/0x160
[<c06814d0>] ? write_sysrq_trigger+0x0/0x50
[<c068150f>] ? write_sysrq_trigger+0x3f/0x50
[<c0569ec4>] ? proc_reg_write+0x64/0xa0
[<c0569e60>] ? proc_reg_write+0x0/0xa0
[<c051de50>] ? vfs_write+0xa0/0x190
[<c051e8d1>] ? sys_write+0x41/0x70
[<c0409adc>] ? syscall_call+0x7/0xb
Code: a0 c0 01 0f b6 41 03 19 d2 f7 d2 83 e2 03 83 e0 cf c1 e2 04 09 d0 88 41 03 f3 c3 90 c7
05 c8 1b 9e c0 01 00 00 0f ae f8 89 f6 <c6> 05 00 00 00 00 01 c3 89 f6 8d bc 27 00 00 00
00 8d 50 d0 83
EIP: [<c068124f>] sysrq_handle_crash+0xf/0x20 SS:ESP 0068:ef4dbf24
CR2: 0000000000000000
```

输入 **help log** 以了解有关命令用法的更多信息。



注意

内核消息缓冲区包含有关系统崩溃的最重要信息，因此始终先转储到 **vmcore-dmesg.txt** 文件。当尝试使完整的 **vmcore** 文件失败时（例如，目标位置上缺少空间），这很有用。默认情况下，**vmcore-dmesg.txt** 位于 **/var/crash/** 目录中。

显示后端

- 若要显示内核堆栈跟踪，可使用 **bt** 命令。

```
crash> bt
PID: 5591 TASK: f196d560 CPU: 2 COMMAND: "bash"
#0 [ef4dbdcc] crash_kexec at c0494922
#1 [ef4dbe20] oops_end at c080e402
#2 [ef4dbe34] no_context at c043089d
#3 [ef4dbe58] bad_area at c0430b26
#4 [ef4dbe6c] do_page_fault at c080fb9b
#5 [ef4dbee4] error_code (via page_fault) at c080d809
EAX: 00000063 EBX: 00000063 ECX: c09e1c8c EDX: 00000000 EBP: 00000000
```

```

DS: 007b   ESI: c0a09ca0 ES: 007b   EDI: 00000286 GS: 00e0
CS: 0060   EIP: c068124f ERR: ffffffff EFLAGS: 00010096
#6 [ef4dbf18] sysrq_handle_crash at c068124f
#7 [ef4dbf24] __handle_sysrq at c0681469
#8 [ef4dbf48] write_sysrq_trigger at c068150a
#9 [ef4dbf54] proc_reg_write at c0569ec2
#10 [ef4dbf74] vfs_write at c051de4e
#11 [ef4dbf94] sys_write at c051e8cc
#12 [ef4dbfb0] system_call at c0409ad5
    EAX: fffffda EBX: 00000001 ECX: b7776000 EDX: 00000002
    DS: 007b   ESI: 00000002 ES: 007b   EDI: b7776000
    SS: 007b   ESP: bfc2088 EBP: bfc20b4 GS: 0033
    CS: 0073   EIP: 00edc416 ERR: 00000004 EFLAGS: 00000246

```

输入 **bt <pid>** 来显示特定进程的回溯追踪，或者输入 **help bt** 以了解有关 **bt** 用法的更多信息。

显示进程状态

- 若要显示系统中进程的状态，可使用 **ps** 命令。

```

crash> ps
  PID  PPID CPU TASK  ST %MEM  VSZ  RSS COMM
>  0    0  0 c09dc560 RU 0.0  0  0 [swapper]
>  0    0  1 f7072030 RU 0.0  0  0 [swapper]
  0    0  2 f70a3a90 RU 0.0  0  0 [swapper]
>  0    0  3 f70ac560 RU 0.0  0  0 [swapper]
  1    0  1 f705ba90 IN 0.0 2828 1424 init
... several lines omitted ...
5566  1  1 f2592560 IN 0.0 12876 784 auditd
5567  1  2 ef427560 IN 0.0 12876 784 auditd
5587 5132  0 f196d030 IN 0.0 11064 3184 sshd
> 5591 5587  2 f196d560 RU 0.0 5084 1648 bash

```

使用 **ps <pid>** 显示单个进程的状态。使用 **help ps** 了解有关 **ps** 用法的更多信息。

显示虚拟内存信息

- 要显示基本虚拟内存信息，请在交互式提示符下键入 **vm** 命令。

```

crash> vm
PID: 5591 TASK: f196d560 CPU: 2 COMMAND: "bash"
MM  PGD  RSS  TOTAL_VM
f19b5900 ef9c6000 1648k 5084k
VMA  START  END  FLAGS FILE
f1bb0310 242000 260000 8000875 /lib/ld-2.12.so
f26af0b8 260000 261000 8100871 /lib/ld-2.12.so
efbc275c 261000 262000 8100873 /lib/ld-2.12.so
efbc2a18 268000 3ed000 8000075 /lib/libc-2.12.so
efbc23d8 3ed000 3ee000 8000070 /lib/libc-2.12.so
efbc2888 3ee000 3f0000 8100071 /lib/libc-2.12.so
efbc2cd4 3f0000 3f1000 8100073 /lib/libc-2.12.so
efbc243c 3f1000 3f4000 100073
efbc28ec 3f6000 3f9000 8000075 /lib/libdl-2.12.so
efbc2568 3f9000 3fa000 8100071 /lib/libdl-2.12.so
efbc2f2c 3fa000 3fb000 8100073 /lib/libdl-2.12.so

```

```
f26af888 7e6000 7fc000 8000075 /lib/libtinfo.so.5.7
f26aff2c 7fc000 7ff000 8100073 /lib/libtinfo.so.5.7
efbc211c d83000 d8f000 8000075 /lib/libnss_files-2.12.so
efbc2504 d8f000 d90000 8100071 /lib/libnss_files-2.12.so
efbc2950 d90000 d91000 8100073 /lib/libnss_files-2.12.so
f26afe00 edc000 edd000 4040075
f1bb0a18 8047000 8118000 8001875 /bin/bash
f1bb01e4 8118000 811d000 8101873 /bin/bash
f1bb0c70 811d000 8122000 100073
f26afae0 9fd9000 9ffa000 100073
... several lines omitted ...
```

使用 **vm <pid>** 显示有关单个特定进程的信息，或使用 **help vm** 了解有关 **vm** 用法的更多信息。

显示打开的文件

- 要显示有关打开文件的信息，请使用 **files** 命令。

```
crash> files
PID: 5591 TASK: f196d560 CPU: 2 COMMAND: "bash"
ROOT: / CWD: /root
FD FILE DENTRY INODE TYPE PATH
0 f734f640 eedc2c6c eecd6048 CHR /pts/0
1 efade5c0 eee14090 f00431d4 REG /proc/sysrq-trigger
2 f734f640 eedc2c6c eecd6048 CHR /pts/0
10 f734f640 eedc2c6c eecd6048 CHR /pts/0
255 f734f640 eedc2c6c eecd6048 CHR /pts/0
```

使用 **files <pid>** 仅显示一个选定进程打开的文件，或者使用 **help files** 来获取有关 **files** 用法的更多信息。

20.4. 使用 KERNEL OOPS ANALYZER

Kernel Oops Analyzer 工具通过将 oops 消息与知识库中已知问题进行比较，分析崩溃转储。

先决条件

- 保护用于馈送内核 Oops 分析器的 oops 消息。

流程

1. 访问内核 Oops 分析器工具。
2. 要诊断内核崩溃问题，请上传 **vmcore** 中生成的内核oops 日志。
 - 或者，您也可以通过提供文本消息或 **vmcore-dmesg.txt** 作为输入来诊断内核崩溃问题。

Option 1: File Input

Choose File No file chosen

Choose and upload the [kernel oops log](#) generated from a vmcore.
Maximum file size for uploaded kernel oops log is 10 MB.

Detect

Option 2: Text Input

Detect Clear

3. 点 **DETECT**，基于 **makedumpfile** 中的信息与已知解决方案比较 oops 消息。

其它资源

- [Kernel Oops 分析器](#) 文章
- [预期系统重启指南](#)

20.5. KDUMP HELPER 工具

Kdump Helper 工具有助于使用提供的信息设置 **kdump**。kdump 帮助程序根据您的偏好生成配置脚本。在服务器中启动并运行该脚本可设置 **kdump** 服务。

其它资源

- [kdump Helper](#)

第 21 章 使用早期 KDUMP 来捕获引导时间崩溃

早期 `kdump` 是 `kdump` 机制的一个特性，如果系统服务启动前，在引导过程早期阶段发生系统或内核崩溃，来捕获 `vmcore` 文件。早期 `kdump` 更早地在内存中加载崩溃内核和崩溃内核的 `initramfs`。

21.1. 什么是早期 KDUMP

在 `kdump` 服务启动前的早期引导阶段，内核崩溃有时会发生，并能够捕获和保存崩溃内核内存的内容。因此，对故障排除很重要的与崩溃相关的重要信息丢失了。要解决这个问题，您可以使用 `early kdump` 功能，其是 `kdump` 服务的一部分。

21.2. 启用早期 KDUMP

`early kdump` 功能设置崩溃内核和初始 RAM 磁盘镜像(`initramfs`)，以便早期加载以捕获早期崩溃的 `vmcore` 信息。这有助于消除丢失早期引导内核崩溃信息的风险。

先决条件

- 一个有效的 RHEL 订阅。
- 包含用于您系统 CPU 架构的 `kexec-tools` 软件包的存储库。
- 实现了 `kdump` 配置和目标要求。如需更多信息，请参阅 [支持的 kdump 配置和目标](#)。

流程

1. 验证 `kdump` 服务是否已启用并活跃：

```
# systemctl is-enabled kdump.service && systemctl is-active kdump.service
enabled
active
```

如果没有启用并运行 `kdump`，请设置所有必要的配置，并验证是否已启用 `kdump` 服务。

2. 使用 `早期 kdump` 功能重建引导内核的 `initramfs` 镜像：

```
# dracut -f --add earlykdump
```

3. 添加 `rd.earlykdump` 内核命令行参数：

```
# grubby --update-kernel=/boot/vmlinuz-$(uname -r) --args="rd.earlykdump"
```

4. 重启系统以反应更改

```
# reboot
```

验证步骤

- 验证 `rd.earlykdump` 是否已成功添加，是否启用了 `early kdump` 功能：

```
# cat /proc/cmdline
BOOT_IMAGE=(hd0,msdos1)/vmlinuz-4.18.0-187.el8.x86_64 root=/dev/mapper/rhel-root ro
```

```
crashkernel=auto resume=/dev/mapper/rhel-swap rd.lvm.lv=rhel/root rd.lvm.lv=rhel/swap  
rhgb quiet rd.earlykdump
```

```
# journalctl -x | grep early-kdump
```

```
Mar 20 15:44:41 redhat dracut-cmdline[304]: early-kdump is enabled.
```

```
Mar 20 15:44:42 redhat dracut-cmdline[304]: kexec: loaded early-kdump kernel
```

其它资源

- [/usr/share/doc/kexec-tools/early-kdump-howto.txt](#) 文件
- [什么是早期 kdump 支持？如何配置它？](#)
- [启用 kdump](#)

第 22 章 使用内核实时修补程序应用补丁

您可以使用 Red Hat Enterprise Linux 内核实时修补解决方案在不重启或者重启任何进程的情况下对运行的内核进行补丁。

使用这个解决方案，系统管理员需要：

- 可以在内核中立即应用重要的安全补丁。
- 不必等待长时间运行的任务完成、关闭或调度停机时间。
- 可以控制系统的正常运行时间，且不会牺牲安全性和稳定性。

请注意，并非所有关键或重要的 CVE 都使用内核实时补丁解决方案来解决。我们的目标是，在应用安全相关的补丁时，尽量减少重启的需要，但无法完全避免重启。有关实时补丁范围的详情，请参阅 [客户门户网站解决方案文章](#)。



警告

内核实时补丁和其它内核子组件之间存在一些不兼容。读
在使用内核实时补丁前，请小心 [kpatch 的限制](#)。



注意

有关内核实时补丁更新支持节奏的详情，请参考：

- [内核实时补丁支持 Cadence 更新](#)
- [内核实时补丁生命周期](#)

22.1. KPATCH 的限制

- **kpatch** 功能不是一个通用内核升级机制。它可用于在无法立即重启系统时应用简单的安全性和程序错误修复更新。
- 不要在载入补丁期间或之后使用 **SystemTap** 或 **kprobe** 工具。在删除此类探测后，补丁可能无法生效。

22.2. 对第三方实时补丁的支持

kpatch 实用程序是红帽通过红帽软件仓库提供的 RPM 模块支持的唯一内核实时补丁程序。红帽不支持任何不是由红帽提供的实时补丁。

如果您需要支持第三方实时补丁所带来的问题，红帽建议您在调查开始时，与需要确定根本原因的任何调查开始时，创建一个与实时补丁供应商案例。如果供应商允许源代码，并且其支持组织能够在向红帽支持升级调查前为其支持组织提供根本原因方面的帮助，则源代码可以被提供。

对于任何使用了第三方补丁程序运行的系统，红帽保留请求用户使用由红帽提供并支持的软件重现问题的权利。如果无法做到这一点，我们需要在测试环境中部署类似的系统和工作负载，而无需应用实时补丁，以确认是否观察到了相同的行为。

有关第三方软件支持政策的更多信息，请参阅[红帽全球支持服务如何处理第三方软件、驱动程序和/或未经认证的硬件/管理程序或虚拟机操作系统？](#)

22.3. 获得内核实时补丁

内核实时补丁功能是作为内核模块（**kmod**）实现的，该模块作为 RPM 软件包提供。

所有客户都可以访问内核实时补丁，这些补丁通过常用的通道提供。但是，在下一个次版本发布后，未订阅延长支持服务的客户将无法访问当前次要版本的新修补程序。例如，具有标准订阅的客户只能在 RHEL 8.3 内核发布前提供 RHEL 8.2 内核补丁。

22.4. 组件内核实时修补

内核实时补丁的组件如下：

内核补丁模块

- 内核实时补丁的交付机制。
- 为内核建补丁的内核模块。
- `patch` 模块包含内核所需修复的代码。
- `patch` 模块使用 **livepatch** 内核子系统注册，并提供要替换的原始功能的信息，并提供与替换功能对应的指针。内核补丁模块以 RPM 的形式提供。
- 命名规则为 **kpatch_<kernel version>_<kpatch version>_<kpatch release>**。名称中"kernel version"部分的点被下划线替代。

kpatch 工具

用于管理补丁模块的命令行工具。

kpatch 服务

multiuser.target 所需的 **systemd** 服务。这个目标会在引导时载入内核补丁模块。

kpatch-dnf 软件包

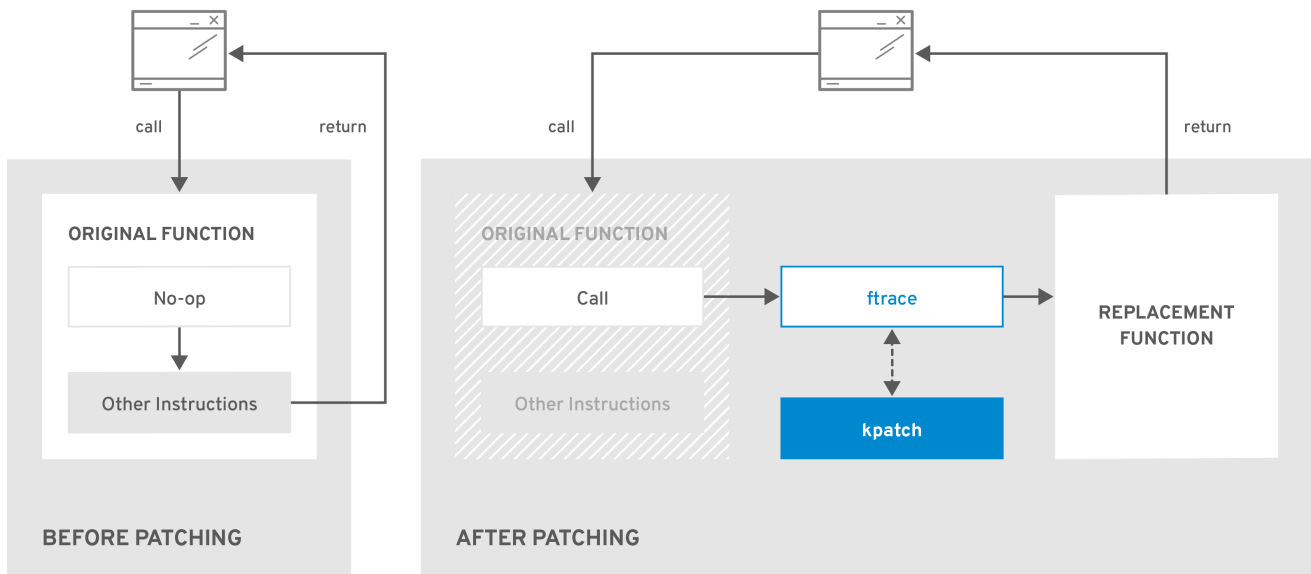
以 RPM 软件包的形式提供的 DNF 插件。此插件管理内核实时补丁的自动订阅。

22.5. 内核实时补丁如何工作

kpatch 内核补丁解决方案使用 **livepatch** 内核子系统将旧功能重定向到新功能。当实时内核补丁应用到系统时，会出现以下情况：

1. 内核补丁模块复制到 **/var/lib/kpatch/** 目录中，并在下次引导时由 **systemd** 注册以重新应用到内核。
2. `kpatch` 模块被加载到正在运行的内核中，新的功能会注册到 **ftrace** 机制中，带有指向新代码内存中位置的指针。
3. 当内核访问补丁的功能时，它将由 **ftrace** 机制重定向，该机制绕过原始功能并将内核重定向到功能补丁版本。

图 22.1. 内核实时补丁如何工作



RHEL_424549_0119

22.6. 将当前安装的内核订阅到实时补丁流

内核补丁模块在 RPM 软件包中提供，具体取决于被修补的内核版本。每个 RPM 软件包将随着时间不断累积更新。

以下流程解释了如何订阅以后为给定内核的所有累积实时补丁更新。因为实时补丁是累计的，所以您无法选择为一个特定的内核部署哪些单独的补丁。



警告

红帽不支持任何适用于红帽支持的系统的第三方实时补丁。

先决条件

- 根权限

流程

1. 另外，还可检查您的内核版本：

```
# uname -r
4.18.0-94.el8.x86_64
```

2. 搜索与内核版本对应的实时补丁软件包：

```
# yum search $(uname -r)
```

3. 安装实时补丁（live patching）软件包：

```
# yum install "kpatch-patch = $(uname -r)"
```

以上命令只为特定内核安装并应用最新的实时补丁。

如果实时补丁软件包的版本是 1-1 或更高版本，则软件包将包含补丁模块。在这种情况下，内核会在安装 live patching 软件包期间自动修补。

内核补丁模块也安装到 `/var/lib/kpatch/` 目录中，供 `systemd` 系统和 `Service Manager` 以后重启时载入。



注意

当给定内核没有可用的实时补丁时，将安装空的实时补丁软件包。空的 live patching 软件包会有一个 0-0 的 `kpatch_version-kpatch_release`，如 `kpatch-patch-4_18_0-94-0-0.el8.x86_64.rpm`。空 RPM 安装会将系统订阅到以后为给定内核提供的所有实时补丁。

验证

- 验证是否所有安装的内核都已打了补丁：

```
# kpatch list
Loaded patch modules:
kpatch_4_18_0_94_1_1 [enabled]

Installed patch modules:
kpatch_4_18_0_94_1_1 (4.18.0-94.el8.x86_64)
...
```

输出显示内核补丁模块已加载到内核，该内核现在已使用 `kpatch-patch-4_18_0-94-1-1.el8.x86_64.rpm` 软件包的最新修复打了补丁。



注意

输入 `kpatch list` 命令不会返回一个空的实时补丁软件包。使用 `rpm -qa | grep kpatch` 命令替代。

```
# rpm -qa | grep kpatch
kpatch-patch-4_18_0-477_21_1-0-0.el8_8.x86_64
kpatch-dnf-0.9.7_0.4-2.el8.noarch
kpatch-0.9.7-2.el8.noarch
```

其它资源

- [kpatch\(1\) 手册页](#)
- [安装软件包](#)

22.7. 自动订阅将来的内核到实时补丁流

您可以使用 `kpatch-dnf` YUM 插件向您的系统订阅内核补丁模块（也称为内核实时补丁）提供的修复。该插件为系统当前使用的任何内核启用自动订阅，以及在以后安装地内核。

先决条件

- 您有 root 权限。

流程

1. (可选) 检查所有安装的内核和您当前运行的内核：

```
# yum list installed | grep kernel
Updating Subscription Management repositories.
Installed Packages
...
kernel-core.x86_64      4.18.0-240.10.1.el8_3    @rhel-8-for-x86_64-baseos-rpms
kernel-core.x86_64      4.18.0-240.15.1.el8_3    @rhel-8-for-x86_64-baseos-rpms
...

# uname -r
4.18.0-240.10.1.el8_3.x86_64
```

2. 安装 `kpatch-dnf` 插件：

```
# yum install kpatch-dnf
```

3. 启用自动订阅内核实时补丁：

```
# yum kpatch auto
Updating Subscription Management repositories.
Last metadata expiration check: 19:10:26 ago on Wed 10 Mar 2021 04:08:06 PM CET.
Dependencies resolved.
=====
Package                Architecture
=====
Installing:
kpatch-patch-4_18_0-240_10_1    x86_64
kpatch-patch-4_18_0-240_15_1    x86_64

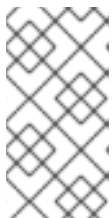
Transaction Summary
=====
Install 2 Packages
...

```

这个命令订阅所有当前安装的内核，以接收内核实时补丁。命令还会为所有安装的内核安装并应用最新的累积实时补丁（如果有）。

将来，当您更新内核时，将在新的内核安装过程中自动安装实时补丁。

内核补丁模块也安装到 `/var/lib/kpatch/` 目录中，供 `systemd` 系统和 `Service Manager` 以后重启时载入。



注意

当给定内核没有可用的实时补丁时，将安装空的实时补丁软件包。空的 live patching 软件包会有一个 0-0 的 `kpatch_version-kpatch_release`，如 `kpatch-patch-4_18_0-240-0-0.el8.x86_64.rpm`。空 RPM 安装会将系统订阅到以后为给定内核提供的所有实时补丁。

验证

- 验证是否所有安装的内核都已打了补丁：

```
# kpatch list
Loaded patch modules:
kpatch_4_18_0_240_10_1_0_1 [enabled]

Installed patch modules:
kpatch_4_18_0_240_10_1_0_1 (4.18.0-240.10.1.el8_3.x86_64)
kpatch_4_18_0_240_15_1_0_2 (4.18.0-240.15.1.el8_3.x86_64)
```

输出显示您正在运行的内核，另一个安装的内核已分别修补了来自 **kpatch-patch-4_18_0-240_10_1-0-1.rpm** 和 **kpatch-patch-4_18_0-240_15_1-0-1.rpm** 软件包的修复。



注意

输入 **kpatch list** 命令不会返回一个空的实时补丁软件包。使用 **rpm -qa | grep kpatch** 命令替代。

```
# rpm -qa | grep kpatch
kpatch-patch-4_18_0-477_21_1-0-0.el8_8.x86_64
kpatch-dnf-0.9.7_0.4-2.el8.noarch
kpatch-0.9.7-2.el8.noarch
```

其它资源

- [kpatch\(1\) 和 dnf-kpatch\(8\)手册页](#)

22.8. 禁用向实时补丁流的自动订阅

当您向内核补丁模块提供的修复订阅系统时，您的订阅是 **自动的**。您可以禁用此功能，从而禁用 **kpatch-patch** 软件包的自动安装。

先决条件

- 您有 root 权限。

流程

1. (可选) 检查所有安装的内核和您当前运行的内核：

```
# yum list installed | grep kernel
Updating Subscription Management repositories.
Installed Packages
...
kernel-core.x86_64      4.18.0-240.10.1.el8_3      @rhel-8-for-x86_64-baseos-rpms
kernel-core.x86_64      4.18.0-240.15.1.el8_3      @rhel-8-for-x86_64-baseos-rpms
...

# uname -r
4.18.0-240.10.1.el8_3.x86_64
```


2. 禁用向内核实时补丁的自动订阅：

```
# yum kpatch manual
Updating Subscription Management repositories.
```

验证步骤

- 您可以检查成功的结果：

```
# yum kpatch status
...
Updating Subscription Management repositories.
Last metadata expiration check: 0:30:41 ago on Tue Jun 14 15:59:26 2022.
Kpatch update setting: manual
```

其它资源

- [kpatch\(1\)](#) 和 [dnf-kpatch\(8\)](#) 手册页

22.9. 更新内核补丁模块

由于内核补丁模块是通过 RPM 软件包交付和应用，更新累积内核补丁模块就如同更新任何其他 RPM 软件包一样。

先决条件

- 系统已订阅实时补丁流，如[将当前安装的内核订阅到实时补丁流](#)中所述。

流程

- 更新至当前内核的新累计版本：

```
# yum update "kpatch-patch = $(uname -r)"
```

以上命令会自动安装并应用所有当前运行的内核可用的更新。包括将来发布的所有在线补丁。

- 另外，更新所有安装的内核补丁模块：

```
# yum update "kpatch-patch"
```



注意

当系统重启到同一内核时，**kpatch.service** systemd 服务会再次对内核进行补丁。

其它资源

- 在 RHEL 中 [配置基本的系统设置](#)

22.10. 删除 LIVE PATCHING 软件包

通过删除实时补丁软件包来禁用 Red Hat Enterprise Linux 内核实时补丁解决方案。

先决条件

- 根权限
- 已安装 live patching 软件包。

流程

1. 选择实时补丁软件包。

```
# yum list installed | grep kpatch-patch
kpatch-patch-4_18_0-94.x86_64    1-1.el8    @@commandline
...
```

上面的输出示例列出了您安装的实时补丁软件包。

2. 删除实时补丁软件包。

```
# yum remove kpatch-patch-4_18_0-94.x86_64
```

删除实时补丁软件包后，内核将保持补丁，直到下次重启为止，但内核补丁模块会从磁盘中删除。将来重启时，对应的内核将不再被修补。

3. 重启您的系统。
4. 验证实时补丁软件包是否已删除。

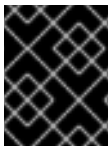
```
# yum list installed | grep kpatch-patch
```

如果软件包已被成功删除，命令不会显示任何输出。

5. (可选) 验证内核实时补丁解决方案是否已禁用。

```
# kpatch list
Loaded patch modules:
```

示例输出显示内核没有补丁，实时补丁解决方案没有激活，因为目前没有加载补丁模块。



重要

目前，红帽不支持在不重启系统的情况下还原实时补丁。如有任何问题，请联系我们的支持团队。

其它资源

- [kpatch \(1\) 手册页](#)
- 在 RHEL 中 [卸载软件包](#)

22.11. 卸载内核补丁模块

防止 Red Hat Enterprise Linux 内核实时补丁解决方案在以后的引导时应用内核补丁模块。

先决条件

- 根权限
- 已安装实时补丁软件包。
- 已安装并载入内核补丁模块。

流程

1. 选择内核补丁模块：

```
# kpatch list
Loaded patch modules:
kpatch_4_18_0_94_1_1 [enabled]

Installed patch modules:
kpatch_4_18_0_94_1_1 (4.18.0-94.el8.x86_64)
...
```

2. 卸载所选的内核补丁模块。

```
# kpatch uninstall kpatch_4_18_0_94_1_1
uninstalling kpatch_4_18_0_94_1_1 (4.18.0-94.el8.x86_64)
```

- 请注意，卸载的内核补丁模块仍然被加载：

```
# kpatch list
Loaded patch modules:
kpatch_4_18_0_94_1_1 [enabled]

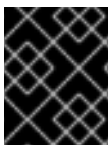
Installed patch modules:
<NO_RESULT>
```

卸载所选模块后，内核将保持补丁，直到下次重启为止，但已从磁盘中删除内核补丁模块。

3. 重启您的系统。
4. (可选) 验证内核补丁模块是否已卸载。

```
# kpatch list
Loaded patch modules:
...
```

以上输出示例显示没有加载或已安装的内核补丁模块，因此没有修补内核，且内核实时补丁解决方案未激活。



重要

目前，红帽不支持在不重启系统的情况下还原实时补丁。如有任何问题，请联系我们的支持团队。

其它资源

- [kpatch \(1\) 手册页](#)

22.12. 禁用 KPATCH.SERVICE

防止 Red Hat Enterprise Linux 内核实时补丁解决方案在以后的引导中全局应用所有内核补丁模块。

先决条件

- 根权限
- 已安装实时补丁软件包。
- 已安装并载入内核补丁模块。

流程

1. 验证 **kpatch.service** 是否已启用。

```
# systemctl is-enabled kpatch.service
enabled
```

2. 禁用 **kpatch.service**。

```
# systemctl disable kpatch.service
Removed /etc/systemd/system/multi-user.target.wants/kpatch.service.
```

- 请注意，应用的内核补丁模块仍然被载入：

```
# kpatch list
Loaded patch modules:
kpatch_4_18_0_94_1_1 [enabled]

Installed patch modules:
kpatch_4_18_0_94_1_1 (4.18.0-94.el8.x86_64)
```

3. 重启您的系统。
4. (可选) 验证 **kpatch.service** 的状态。

```
# systemctl status kpatch.service
● kpatch.service - "Apply kpatch kernel patches"
  Loaded: loaded (/usr/lib/systemd/system/kpatch.service; disabled; vendor preset: disabled)
  Active: inactive (dead)
```

示例输出测试 **kpatch.service** 已被禁用且没有在运行。因此，内核实时修补解决方案不活跃。

5. 验证内核补丁模块是否已卸载。

```
# kpatch list
Loaded patch modules:
<NO_RESULT>

Installed patch modules:
kpatch_4_18_0_94_1_1 (4.18.0-94.el8.x86_64)
```

上面的示例输出显示内核补丁模块仍处于安装状态，但没有修补内核。



重要

目前，红帽不支持在不重启系统的情况下还原实时补丁。如有任何问题，请联系我们的支持团队。

其它资源

- **kpatch (1)** 手册页 [管理 systemd](#)

第 23 章 使用控制组群为应用程序设置系统资源限制

使用控制组(**cgroups**)内核功能，您可以控制应用程序的资源使用情况来更有效地使用它们。

您可以为以下任务使用 **cgroups**：

- 为系统资源分配设置限制。
- 将硬件资源优先分配给特定的进程。
- 防止某些进程获取硬件资源。

23.1. 控制组简介

使用 *控制组* Linux 内核功能，您可以将进程组织为按层排序的组 - **cgroups**。您可以通过为 **cgroup** 虚拟文件系统提供结构来定义层次结构（控制组树），默认挂载到 **/sys/fs/cgroup/** 目录。

systemd 服务管理器使用 **cgroups** 来组织它管理的所有单元和服务。您可以通过创建和删除 **/sys/fs/cgroup/** 目录中的子目录来手动管理 **cgroups** 的层次结构。

然后，内核中的资源控制器通过限制、优先处理或分配这些进程的系统资源来在 **cgroups** 中修改进程的行为。这些资源包括以下内容：

- CPU 时间
- 内存
- 网络带宽
- 这些资源的组合

cgroups 的主要用例是聚合系统进程，并在应用程序和用户之间划分硬件资源。这样可以提高环境的效率、稳定性和安全性。

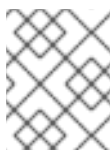
控制组群版本 1

控制组版本 1 (cgroups-v1) 提供按资源控制器层次结构。这意味着每个资源（如 CPU、内存或 I/O）都有自己的控制组层次结构。您可以组合不同的控制组层次结构，使一个控制器可以与另一个控制器协调管理各自的资源。但是，当两个控制器属于不同的进程层次结构时，合适的协调会被限制。

cgroups-v1 控制器在很长的时间跨度开发，因此其控制文件的行为和命名不一致。

控制组群版本 2

控制组版本 2 (cgroups-v2) 提供单一控制组层次结构，用于挂载所有资源控制器。控制文件行为和命名在不同控制器之间保持一致。



注意

RHEL 8.2 及更新的版本完全支持 **cgroups-v2**。如需更多信息，请参阅 [RHEL 8 现在完全支持控制组群 v2](#)。

其它资源

- [内核资源控制器简介](#)

- [cgroups \(7\) 手册页](#)
- [控制组群中的 systemd 角色](#)

23.2. 内核资源控制器简介

内核资源控制器启用控制组的功能。RHEL 8 支持 *控制组群版本 1 (cgroups-v1)* 和 *控制组版本 2 (cgroups-v2)* 的各种控制器。

资源控制器也称为控制组子系统，是一个代表单一资源的内核子系统，如 CPU 时间、内存、网络带宽或磁盘 I/O。Linux 内核提供由 **systemd** 服务管理器自动挂载的一系列资源控制器。您可以在 `/proc/cgroups` 文件中找到当前挂载的资源控制器的列表。

cgroups-v1 提供的控制器：

blkio

设置对块设备的输入/输出访问的限制。

cpu

调整控制组任务的完全公平调度程序(CFS)的参数。**cpu** 控制器与 **cpuacct** 控制器一起挂载在同一挂载上。

cpuacct

创建控制组群中任务所使用的有关 CPU 资源的自动报告。**cpuacct** 控制器与 **cpu** 控制器一起挂载在同一挂载上。

cpuset

将控制组任务限制为仅在指定 CPU 子集上运行，并指示任务仅使用指定内存节点上的内存。

devices

控制控制组群中任务对设备的访问。

freezer

暂停或恢复控制组中的任务。

内存

设置控制组中任务对内存使用的限制，并对这些任务使用的内存资源生成自动报告。

net_cls

使用类标识符(**classid**)标记网络数据包，使 Linux 流量控制器(**tc** 命令)能够识别来自特定控制组任务的数据包。**net_cls** 子系统 **net_filter** (iptables) 也可使用此标签对此类数据包执行操作。**net_filter** 使用防火墙标识符(**fwid**)标记网络套接字，它允许 Linux 防火墙识别来自特定控制组任务的数据包（通过使用 **iptables** 命令）。

net_prio

设置网络流量的优先级。

pids

为控制组中的多个进程及其子进程设置限制。

perf_event

通过 **perf** 性能监控和报告工具对监控的任务进行分组。

rdma

对控制组群中远程直接内存访问/InfiniBand 特定资源设置限制。

hugetlb

可用于限制控制组中按任务的大量虚拟内存页的使用率。

cgroups-v2 提供的控制器：**io**

设置对块设备的输入/输出访问的限制。

内存

设置控制组中任务对内存使用的限制，并对这些任务使用的内存资源生成自动报告。

pids

为控制组中的多个进程及其子进程设置限制。

rdma

对控制组群中远程直接内存访问/InfiniBand 特定资源设置限制。

cpu

为控制组的任务调整完全公平调度程序(CFS)的参数，并创建控制组中任务所使用的 CPU 资源的自动报告。

cpuset

将控制组任务限制为仅在指定 CPU 子集上运行，并指示任务仅使用指定内存节点上的内存。仅支持具有新分区功能的核心功能(`cpus{,.effective}`, `mems{,.effective}`)。

perf_event

通过 **perf** 性能监控和报告工具对监控的任务进行分组。**perf_event** 在 v2 层次结构上自动启用。

**重要**

资源控制器可以在 **cgroups-v1** 层次结构或 **cgroups-v2** 层次结构中使用，不能同时在两者中使用。

其它资源

- **cgroups (7)** 手册页
- `/usr/share/doc/kernel-doc-<kernel_version>/Documentation/cgroups-v1/` 目录中的文档（安装 **kernel-doc** 软件包之后）。

23.3. 命名空间简介

命名空间是整理和识别软件对象的最重要的方法之一。

命名空间将全局系统资源（例如挂载点、网络设备或主机名）封装在抽象中，使名称空间中的进程看起来拥有自己的全局资源的隔离实例。使用命名空间的最常见技术是容器。

对特定全局资源的更改仅对该命名空间中的进程可见，不影响系统或其他命名空间的其余部分。

要检查进程所属的命名空间，您可以在 `/proc/<PID>/ns/` 目录中检查符号链接。

表 23.1. 支持的命名空间以及它们隔离的资源：

命名空间	Isolates
Mount	挂载点
UTS	主机名和 NIS 域名

命名空间	Isolates
IPC	系统 V IPC, POSIX 消息队列
PID	进程 ID
Network	网络设备、堆栈、端口等
User	用户和组群 ID
Control groups	控制组群根目录

其它资源

- [namespaces \(7\)](#) 和 [cgroup_namespaces \(7\)](#) 手册页
- [控制组简介](#)

23.4. 使用 CGROUPS-V1 为应用程序设置 CPU 限制

要使用 *控制组版本 1* (`cgroups-v1`) 配置对应用程序的 CPU 限制，请使用 `/sys/fs/` 虚拟文件系统。

先决条件

- 您有 root 权限。
- 您有一个应用程序,您想限制其 CPU 消耗。
- 您确认已挂载了 `cgroups-v1` 控制器：

```
# mount -l | grep cgroup
tmpfs on /sys/fs/cgroup type tmpfs (ro,nosuid,nodev,noexec,seclabel,mode=755)
cgroup on /sys/fs/cgroup/systemd type cgroup
(rw,nosuid,nodev,noexec,relatime,seclabel,xattr,release_agent=/usr/lib/systemd/systemd-
cgroups-agent,name=systemd)
cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup
(rw,nosuid,nodev,noexec,relatime,seclabel,cpu,cpuacct)
cgroup on /sys/fs/cgroup/perf_event type cgroup
(rw,nosuid,nodev,noexec,relatime,seclabel,perf_event)
cgroup on /sys/fs/cgroup/pids type cgroup (rw,nosuid,nodev,noexec,relatime,seclabel,pids)
...
```

流程

1. 识别您要限制 CPU 消耗的应用程序的进程 ID (PID)：

```
# top
top - 11:34:09 up 11 min, 1 user, load average: 0.51, 0.27, 0.22
Tasks: 267 total, 3 running, 264 sleeping, 0 stopped, 0 zombie
%Cpu(s): 49.0 us, 3.3 sy, 0.0 ni, 47.5 id, 0.0 wa, 0.2 hi, 0.0 si, 0.0 st
MiB Mem : 1826.8 total, 303.4 free, 1046.8 used, 476.5 buff/cache
```

```
MiB Swap: 1536.0 total, 1396.0 free, 140.0 used. 616.4 avail Mem
```

```

PID USER   PR NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
6955 root    20  0 228440 1752 1472 R 99.3  0.1   0:32.71 sha1sum
5760 jdoe    20  0 3603868 205188 64196 S  3.7 11.0   0:17.19 gnome-shell
6448 jdoe    20  0 743648 30640 19488 S  0.7  1.6   0:02.73 gnome-terminal-
 1 root    20  0 245300 6568 4116 S  0.3  0.4   0:01.87 systemd
505 root    20  0  0  0  0 l 0.3  0.0   0:00.75 kworker/u4:4-events_unbound
...

```

这个 **top** 程序的示例输出显示，**PID 6955** 的说明性应用程序 **sha1sum** 消耗了大量 CPU 资源。

2. 在 **cpu** 资源控制器目录中创建子目录：

```
# mkdir /sys/fs/cgroup/cpu/Example/
```

此目录代表控制组，您可以在其中放置特定进程，并向进程应用某些 CPU 限制。同时，目录中将创建多个 **cgroups-v1** 接口文件和 **cpu** 特定于控制器的文件。

3. 可选：检查新创建的控制组：

```

# ll /sys/fs/cgroup/cpu/Example/
-rw-r--r--. 1 root root 0 Mar 11 11:42 cgroup.clone_children
-rw-r--r--. 1 root root 0 Mar 11 11:42 cgroup.procs
-r--r--r--. 1 root root 0 Mar 11 11:42 cpuacct.stat
-rw-r--r--. 1 root root 0 Mar 11 11:42 cpuacct.usage
-r--r--r--. 1 root root 0 Mar 11 11:42 cpuacct.usage_all
-r--r--r--. 1 root root 0 Mar 11 11:42 cpuacct.usage_percpu
-r--r--r--. 1 root root 0 Mar 11 11:42 cpuacct.usage_percpu_sys
-r--r--r--. 1 root root 0 Mar 11 11:42 cpuacct.usage_percpu_user
-r--r--r--. 1 root root 0 Mar 11 11:42 cpuacct.usage_sys
-r--r--r--. 1 root root 0 Mar 11 11:42 cpuacct.usage_user
-rw-r--r--. 1 root root 0 Mar 11 11:42 cpu.cfs_period_us
-rw-r--r--. 1 root root 0 Mar 11 11:42 cpu.cfs_quota_us
-rw-r--r--. 1 root root 0 Mar 11 11:42 cpu.rt_period_us
-rw-r--r--. 1 root root 0 Mar 11 11:42 cpu.rt_runtime_us
-rw-r--r--. 1 root root 0 Mar 11 11:42 cpu.shares
-r--r--r--. 1 root root 0 Mar 11 11:42 cpu.stat
-rw-r--r--. 1 root root 0 Mar 11 11:42 notify_on_release
-rw-r--r--. 1 root root 0 Mar 11 11:42 tasks

```

这个示例输出显示文件，如 **cpuacct.usage**、**cpu.cfs_period_us**，它们代表特定的配置和/或限值，可以为 **Example** 控制组中的进程设置。请注意，对应的文件名以它们所属的控制组控制器的名称为前缀。

默认情况下，新创建的控制组继承对系统整个 CPU 资源的访问权限，且无限制。

4. 为控制组群配置 CPU 限制：

```
# echo "1000000" > /sys/fs/cgroup/cpu/Example/cpu.cfs_period_us
# echo "200000" > /sys/fs/cgroup/cpu/Example/cpu.cfs_quota_us
```

- **cpu.cfs_period_us** 文件表示以微秒为单位（这里表示为"us"）的时段，用于控制组对 CPU 资源的访问权限应重新分配的频率。上限为 1 000 000 微秒，下限为 1 000 微秒。

- **cpu.cfs_quota_us** 文件表示以微秒为单位的总时间量，控制组中的所有进程都可以在一个期间（如 **cpu.cfs_period_us** 定义）。当控制组中的进程在单个期间内使用完配额指定的所有时间时，在周期的剩余时间内，它们将被节流，并且不允许在下一个期间前运行。较低限制为 1 000 微秒。
上面的示例命令设定 CPU 时间限值，使得 **Example** 控制组中的所有进程仅能每 1 秒（**cpu.cfs_quota_us** 定义）每 1 秒（由 **cpu.cfs_period_us** 定义）运行 0.2 秒。

5. 可选：验证限制：

```
# cat /sys/fs/cgroup/cpu/Example/cpu.cfs_period_us
/sys/fs/cgroup/cpu/Example/cpu.cfs_quota_us
1000000
200000
```

6. 将应用程序的 PID 添加到 **Example** 控制组群中：

```
# echo "6955" > /sys/fs/cgroup/cpu/Example/cgroup.procs
```

此命令确保特定应用成为 **Example** 控制组的一员，因此不超过为 **Example** 控制组配置的 CPU 限值。PID 应该代表系统中的一个已存在的进程。这里的 PID 6955 分配给进程 **sha1sum /dev/zero &**，用于说明 **cpu** 控制器的用例。

验证

1. 验证应用程序是否在指定的控制组群中运行：

```
# cat /proc/6955/cgroup
12:cpuset:/
11:hugetlb:/
10:net_cls,net_prio:/
9:memory:/user.slice/user-1000.slice/user@1000.service
8:devices:/user.slice
7:blkio:/
6:freezer:/
5:rdma:/
4:pids:/user.slice/user-1000.slice/user@1000.service
3:perf_event:/
2:cpu,cpuacct:/Example
1:name=systemd:/user.slice/user-1000.slice/user@1000.service/gnome-terminal-server.service
```

此示例输出显示在 **Example** 控制组中运行的所需应用程序的进程，后者将 CPU 限制应用到应用程序的进程。

2. 确定节流应用程序的当前 CPU 消耗：

```
# top
top - 12:28:42 up 1:06, 1 user, load average: 1.02, 1.02, 1.00
Tasks: 266 total, 6 running, 260 sleeping, 0 stopped, 0 zombie
%Cpu(s): 11.0 us, 1.2 sy, 0.0 ni, 87.5 id, 0.0 wa, 0.2 hi, 0.0 si, 0.2 st
MiB Mem : 1826.8 total, 287.1 free, 1054.4 used, 485.3 buff/cache
MiB Swap: 1536.0 total, 1396.7 free, 139.2 used. 608.3 avail Mem

  PID USER   PR NI  VIRT  RES  SHR S %CPU %MEM  TIME+ COMMAND
 6955 root    20  0 228440 1752 1472 R  20.6  0.1 47:11.43 sha1sum
```

```
5760 jdoe    20 0 3604956 208832 65316 R  2.3 11.2  0:43.50 gnome-shell
6448 jdoe    20 0 743836 31736 19488 S  0.7  1.7  0:08.25 gnome-terminal-
505 root     20 0    0    0    0 | 0.3 0.0  0:03.39 kworker/u4:4-events_unbound
4217 root     20 0 74192 1612 1320 S  0.3  0.1  0:01.19 spice-vdagentd
...
```

请注意，**PID 6955** 的 CPU 消耗从 99% 减少到 20%。



注意

`cpu.cfs_period_us` 和 `cpu.cfs_quota_us` 的 `cgroups-v2` 对应项是 `cpu.max` 文件。`cpu.max` 文件可以通过 `cpu` 控制器获得。

其它资源

- [了解控制组群](#)
- [内核资源控制器](#)
- [cgroups\(7\)、sysfs\(5\) 手册页](#)

第 24 章 使用 CGROUPS-V2 控制应用程序的 CPU 时间分布

有些应用程序占用太多的 CPU 时间，这可能会对您环境的整体健康状况造成负面影响。您可以将应用程序置于 *控制组版本 2 (cgroups-v2)* 中，并为这些控制组配置 CPU 限制。因此，您可以控制应用程序的 CPU 消耗。

关于如何控制分配给控制组群的 CPU 时间分布，用户有两种方法：

- 设置 CPU 带宽（编辑 `cpu.max` 控制器文件）
- 设置 CPU 权重（编辑 `cpu.weight` 控制器文件）

24.1. 挂载 CGROUPS-V2

在启动过程中，RHEL 8 默认挂载 `cgroup-v1` 虚拟文件系统。要利用 `cgroup-v2` 功能限制应用程序的资源，请手动配置系统。

先决条件

- 您有 root 权限。

流程

1. 通过 `systemd` 系统和 `服务管理器` 系统，在系统引导期间将系统配置为默认挂载 `cgroups-v2`：

```
# grubby --update-kernel=/boot/vmlinuz-$(uname -r) --
args="systemd.unified_cgroup_hierarchy=1"
```

这会向当前引导条目中添加必要的内核命令行参数。

向所有内核引导条目中添加 `systemd.unified_cgroup_hierarchy=1` 参数：

```
# grubby --update-kernel=ALL --args="systemd.unified_cgroup_hierarchy=1"
```

2. 重启系统以使更改生效。

验证步骤

1. （可选）验证是否 `cgroups-v2` 文件系统已挂载：

```
# mount -l | grep cgroup
cgroup2 on /sys/fs/cgroup type cgroup2
(rw,nosuid,nodev,noexec,relatime,seclabel,nsdelegate)
```

`cgroups-v2` 文件系统已成功挂载到 `/sys/fs/cgroup/` 目录。

2. （可选）检查 `/sys/fs/cgroup/` 目录的内容：

```
# ll /sys/fs/cgroup/
-r--r--r--. 1 root root 0 Apr 29 12:03 cgroup.controllers
-rw-r--r--. 1 root root 0 Apr 29 12:03 cgroup.max.depth
-rw-r--r--. 1 root root 0 Apr 29 12:03 cgroup.max.descendants
-rw-r--r--. 1 root root 0 Apr 29 12:03 cgroup.procs
-r--r--r--. 1 root root 0 Apr 29 12:03 cgroup.stat
```

```

-rw-r--r--. 1 root root 0 Apr 29 12:18 cgroup.subtree_control
-rw-r--r--. 1 root root 0 Apr 29 12:03 cgroup.threads
-rw-r--r--. 1 root root 0 Apr 29 12:03 cpu.pressure
-r--r--r--. 1 root root 0 Apr 29 12:03 cpuset.cpus.effective
-r--r--r--. 1 root root 0 Apr 29 12:03 cpuset.mems.effective
-r--r--r--. 1 root root 0 Apr 29 12:03 cpu.stat
drwxr-xr-x. 2 root root 0 Apr 29 12:03 init.scope
-rw-r--r--. 1 root root 0 Apr 29 12:03 io.pressure
-r--r--r--. 1 root root 0 Apr 29 12:03 io.stat
-rw-r--r--. 1 root root 0 Apr 29 12:03 memory.pressure
-r--r--r--. 1 root root 0 Apr 29 12:03 memory.stat
drwxr-xr-x. 69 root root 0 Apr 29 12:03 system.slice
drwxr-xr-x. 3 root root 0 Apr 29 12:18 user.slice

```

`/sys/fs/cgroup/` 目录也称为 *根控制组*，默认情况下，包含接口文件（以 `cgroup` 开头）和特定于控制器的文件，如 `cpuset.cpus.effective`。此外，还有一些与 `systemd` 相关的目录，如 `/sys/fs/cgroup/init.scope`、`/sys/fs/cgroup/system.slice` 和 `/sys/fs/cgroup/user.slice`。

其它资源

- [了解控制组群](#)
- [内核资源控制器](#)
- [cgroups\(7\)、sysfs\(5\) 手册页](#)

24.2. 为 CPU 时间分布准备 CGROUP

要控制应用程序的 CPU 消耗，您需要启用特定的 CPU 控制器，并创建一个专用的控制组。建议您在 `/sys/fs/cgroup/` 根控制组群中至少创建两级子控制组，以更好地保持 `cgroup` 文件的组织清晰性。

先决条件

- 您有 `root` 权限。
- 您已确定了要控制的进程的 PID。
- 您已挂载了 `cgroups-v2` 文件系统。如需更多信息，请参阅 [挂载 cgroups-v2](#)。

流程

1. 识别您要限制其 CPU 消耗的应用程序的进程 ID(PID)：

```

# top
Tasks: 104 total, 3 running, 101 sleeping, 0 stopped, 0 zombie
%Cpu(s): 17.6 us, 81.6 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.8 hi, 0.0 si, 0.0 st
MiB Mem : 3737.4 total, 3312.7 free, 133.3 used, 291.4 buff/cache
MiB Swap: 4060.0 total, 4060.0 free, 0.0 used. 3376.1 avail Mem

  PID USER   PR NI  VIRT  RES  SHR S %CPU %MEM  TIME+ COMMAND
 34578 root    20  0 18720 1756 1468 R  99.0  0.0  0:31.09 sha1sum
 34579 root    20  0 18720 1772 1480 R  99.0  0.0  0:30.54 sha1sum
    1 root    20  0 186192 13940 9500 S  0.0  0.4  0:01.60 systemd
    2 root    20  0    0    0    0 S  0.0  0.0  0:00.01 kthreadd

```

```

3 root    0 -20    0  0  0 | 0.0  0.0  0:00.00 rcu_gp
4 root    0 -20    0  0  0 | 0.0  0.0  0:00.00 rcu_par_gp
...

```

示例输出显示 **PID 34578** 和 **34579**（**sha1sum** 的两个说明性应用程序）消耗了大量资源，即 CPU。这两个应用程序都是用于演示 **cgroups-v2** 功能的应用示例。

- 验证 **/sys/fs/cgroup/cgroup.controllers** 文件中是否提供了 **cpu** 和 **cpuset** 控制器：

```

# cat /sys/fs/cgroup/cgroup.controllers
cpuset cpu io memory hugetlb pids rdma

```

- 启用与 CPU 相关的控制器：

```

# echo "+cpu" >> /sys/fs/cgroup/cgroup.subtree_control
# echo "+cpuset" >> /sys/fs/cgroup/cgroup.subtree_control

```

这些命令为 **/sys/fs/cgroup/** 根控制组的直接子组启用了 **cpu** 和 **cpuset** 控制器。子组是您可以指定进程，并根据您的标准对每个进程应用控制检查的地方。

用户可以在任意级别读取 **cgroup.subtree_control** 文件的内容，以了解即时子组中哪些控制器可用于启用。



注意

默认情况下，根控制组中的 **/sys/fs/cgroup/cgroup.subtree_control** 文件包含 **memory** 和 **pids** 控制器。

- 创建 **/sys/fs/cgroup/Example/** 目录：

```

# mkdir /sys/fs/cgroup/Example/

```

/sys/fs/cgroup/Example/ 目录定义了一个子组。此外，上一步为这个子组启用了 **cpu** 和 **cpuset** 控制器。

创建 **/sys/fs/cgroup/Example/** 目录时，一些 **cgroups-v2** 接口文件以及 **cpu** 和 **cpuset** 特定于控制器的文件也会在目录中自动创建。**/sys/fs/cgroup/Example/** 目录还包含 **memory** 和 **pids** 控制器的特定于控制器的文件。

- 可选，检查新创建的子控制组：

```

# ll /sys/fs/cgroup/Example/
-r--r--r--. 1 root root 0 Jun  1 10:33 cgroup.controllers
-r--r--r--. 1 root root 0 Jun  1 10:33 cgroup.events
-rw-r--r--. 1 root root 0 Jun  1 10:33 cgroup.freeze
-rw-r--r--. 1 root root 0 Jun  1 10:33 cgroup.max.depth
-rw-r--r--. 1 root root 0 Jun  1 10:33 cgroup.max.descendants
-rw-r--r--. 1 root root 0 Jun  1 10:33 cgroup.procs
-r--r--r--. 1 root root 0 Jun  1 10:33 cgroup.stat
-rw-r--r--. 1 root root 0 Jun  1 10:33 cgroup.subtree_control
...
-rw-r--r--. 1 root root 0 Jun  1 10:33 cpuset.cpus
-r--r--r--. 1 root root 0 Jun  1 10:33 cpuset.cpus.effective
-rw-r--r--. 1 root root 0 Jun  1 10:33 cpuset.cpus.partition

```

```
-rw-r--r--. 1 root root 0 Jun  1 10:33 cpuset.mems
-r--r--r--. 1 root root 0 Jun  1 10:33 cpuset.mems.effective
-r--r--r--. 1 root root 0 Jun  1 10:33 cpu.stat
-rw-r--r--. 1 root root 0 Jun  1 10:33 cpu.weight
-rw-r--r--. 1 root root 0 Jun  1 10:33 cpu.weight.nice
...
-r--r--r--. 1 root root 0 Jun  1 10:33 memory.events.local
-rw-r--r--. 1 root root 0 Jun  1 10:33 memory.high
-rw-r--r--. 1 root root 0 Jun  1 10:33 memory.low
...
-r--r--r--. 1 root root 0 Jun  1 10:33 pids.current
-r--r--r--. 1 root root 0 Jun  1 10:33 pids.events
-rw-r--r--. 1 root root 0 Jun  1 10:33 pids.max
```

示例输出显示 **cpuset.cpus** 和 **cpu.max** 等文件。这些文件特定于 **cpuset** 和 **cpu** 控制器。使用 **/sys/fs/cgroup/cgroup.subtree_control** 文件，为根的 (**/sys/fs/cgroup/**)直接子控制组手动启用 **cpuset** 和 **cpu** 控制器。

目录还包含通用 **cgroup** 控制接口文件，如 **cgroup.procs** 或 **cgroup.controllers**，它们对所有控制组都通用，无论启用的控制器是什么。

memory.high 和 **pids.max** 等文件与 **memory** 和 **pids** 控制器有关，它们位于根控制组中 (**/sys/fs/cgroup/**)，默认情况下总是启用的。

默认情况下，新创建的子组继承了对系统所有 CPU 和内存资源的访问权限，没有任何限制。

6. 启用 **/sys/fs/cgroup/Example/** 中与 CPU 相关的控制器，以获取仅与 CPU 相关的控制器：

```
# echo "+cpu" >> /sys/fs/cgroup/Example/cgroup.subtree_control
# echo "+cpuset" >> /sys/fs/cgroup/Example/cgroup.subtree_control
```

这些命令确保直接子控制组将 *只能* 有与控制 CPU 时间相关的控制器，而不是 **memory** 或 **pids** 控制器。

7. 创建 **/sys/fs/cgroup/Example/tasks/** 目录：

```
# mkdir /sys/fs/cgroup/Example/tasks/
```

/sys/fs/cgroup/Example/tasks/ 目录定义了一个子组，以及只与 **cpu** 和 **cpuset** 控制器相关的文件。

8. 另外，还可检查其他子控制组：

```
# ll /sys/fs/cgroup/Example/tasks
-r--r--r--. 1 root root 0 Jun  1 11:45 cgroup.controllers
-r--r--r--. 1 root root 0 Jun  1 11:45 cgroup.events
-rw-r--r--. 1 root root 0 Jun  1 11:45 cgroup.freeze
-rw-r--r--. 1 root root 0 Jun  1 11:45 cgroup.max.depth
-rw-r--r--. 1 root root 0 Jun  1 11:45 cgroup.max.descendants
-rw-r--r--. 1 root root 0 Jun  1 11:45 cgroup.procs
-r--r--r--. 1 root root 0 Jun  1 11:45 cgroup.stat
-rw-r--r--. 1 root root 0 Jun  1 11:45 cgroup.subtree_control
-rw-r--r--. 1 root root 0 Jun  1 11:45 cgroup.threads
-rw-r--r--. 1 root root 0 Jun  1 11:45 cgroup.type
-rw-r--r--. 1 root root 0 Jun  1 11:45 cpu.max
-rw-r--r--. 1 root root 0 Jun  1 11:45 cpu.pressure
```



```
-rw-r--r--. 1 root root 0 Jun  1 11:45 cpuset.cpus
-r--r--r--. 1 root root 0 Jun  1 11:45 cpuset.cpus.effective
-rw-r--r--. 1 root root 0 Jun  1 11:45 cpuset.cpus.partition
-rw-r--r--. 1 root root 0 Jun  1 11:45 cpuset.mems
-r--r--r--. 1 root root 0 Jun  1 11:45 cpuset.mems.effective
-r--r--r--. 1 root root 0 Jun  1 11:45 cpu.stat
-rw-r--r--. 1 root root 0 Jun  1 11:45 cpu.weight
-rw-r--r--. 1 root root 0 Jun  1 11:45 cpu.weight.nice
-rw-r--r--. 1 root root 0 Jun  1 11:45 io.pressure
-rw-r--r--. 1 root root 0 Jun  1 11:45 memory.pressure
```

9. 确保您要控制 CPU 时间的进程在同一 CPU 上竞争：

```
# echo "1" > /sys/fs/cgroup/Example/tasks/cpuset.cpus
```

以上命令可确保您将放在 **Example/tasks** 子控制组中的进程在同一 CPU 上竞争。此设置对于要激活的 **cpu** 控制器非常重要。



重要

cpu 控制器只有在相关子控制组至少有 2 个在单个 CPU 上竞争时间的进程时，才会被激活。

验证步骤

1. 可选：确保为直接子 cgroups 启用了与 CPU 相关的控制器：

```
# cat /sys/fs/cgroup/cgroup.subtree_control
/sys/fs/cgroup/Example/cgroup.subtree_control
cpuset cpu memory pids
cpuset cpu
```

2. 可选：确保您要控制 CPU 时间的进程在同一 CPU 上竞争：

```
# cat /sys/fs/cgroup/Example/tasks/cpuset.cpus
1
```

其它资源

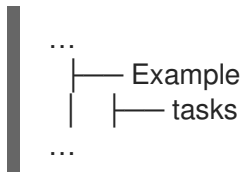
- [什么是控制组](#)
- [什么是内核资源控制器](#)
- [挂载 cgroups-v2](#)
- [cgroups\(7\)、sysfs\(5\) 手册页](#)

24.3. 通过调整 CPU 带宽来控制应用程序的 CPU 时间分布

您需要为 **cpu** 控制器的相关文件分配值，以控制分发到特定 cgroup 树下应用程序的 CPU 时间。

先决条件

- 您有 root 权限。
- 您至少有两个用于控制 CPU 时间分发的应用程序。
- 您确保相关的应用程序在同一 CPU 上竞争 CPU 时间，如 [为 CPU 时间分布准备 cgroup](#) 中所述。
- 您已挂载了 **cgroups-v2** 文件系统，如 [挂载 cgroups-v2](#) 中所述。
- 您在父控制组和子控制组中启用了 **cpu** 和 **cpuset** 控制器，如 [为 CPU 时间分布准备 cgroup](#) 中所述。
- 您在 **/sys/fs/cgroup/** 根控制组中创建了两级子控制组，如下例所示：



流程

1. 配置 CPU 带宽以在控制组群内实现资源限制：

```
# echo "200000 1000000" > /sys/fs/cgroup/Example/tasks/cpu.max
```

第一个值是允许的时间配额（以微秒为单位），用于子组中的所有进程可以在一段时间内运行。第二个值指定时间段的长度。

在一段时间内，当控制组中的进程共同耗尽此配额指定的时间时，它们会在该时间段的剩余时间内节流，并且直到下一个时间段才允许运行。

此命令设置 CPU 时间分布控制，以便 **/sys/fs/cgroup/Example/tasks** 子组中的所有进程每 1 秒只能在 CPU 上运行 0.2 秒。也就是说，每秒的五分之一。

2. 另外，验证时间配额：

```
# cat /sys/fs/cgroup/Example/tasks/cpu.max
200000 1000000
```

3. 将应用的 PID 添加到 **Example/tasks** 子组中：

```
# echo "34578" > /sys/fs/cgroup/Example/tasks/cgroup.procs
# echo "34579" > /sys/fs/cgroup/Example/tasks/cgroup.procs
```

示例命令确保所需的应用程序成为 **Example/tasks** 子组的成员，并且不超过为此子组配置的 CPU 时间分布。

验证步骤

1. 验证应用程序是否在指定的控制组群中运行：

```
# cat /proc/34578/cgroup /proc/34579/cgroup
0::/Example/tasks
0::/Example/tasks
```

上面的输出显示了在 **Example/tasks** 子组中运行的指定应用程序的进程。

2. 检查节流应用程序的当前 CPU 消耗：

```
# top
top - 11:13:53 up 23:10, 1 user, load average: 0.26, 1.33, 1.66
Tasks: 104 total, 3 running, 101 sleeping, 0 stopped, 0 zombie
%Cpu(s): 3.0 us, 7.0 sy, 0.0 ni, 89.5 id, 0.0 wa, 0.2 hi, 0.2 si, 0.2 st
MiB Mem : 3737.4 total, 3312.6 free, 133.4 used, 291.4 buff/cache
MiB Swap: 4060.0 total, 4060.0 free, 0.0 used. 3376.0 avail Mem

  PID USER  PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 34578 root   20   0 18720 1756 1468 R  10.0  0.0 37:36.13 sha1sum
 34579 root   20   0 18720 1772 1480 R  10.0  0.0 37:41.22 sha1sum
    1 root   20   0 186192 13940 9500 S  0.0  0.4 0:01.60 systemd
    2 root   20   0    0    0    0 S  0.0  0.0 0:00.01 kthreadd
    3 root    0 -20    0    0    0 I  0.0  0.0 0:00.00 rcu_gp
    4 root    0 -20    0    0    0 I  0.0  0.0 0:00.00 rcu_par_gp
...

```

请注意，**PID 34578** 和 **PID 34579** 的 CPU 消耗已减少到 10%。**Example/tasks** 子组将其进程控制到 CPU 时间的 20%。由于控制组中有 2 个进程，因此每个进程可以使用 10% 的 CPU 时间。

其它资源

- [了解控制组群](#)
- [内核资源控制器](#)
- [挂载 cgroups-v2](#)
- [为 CPU 时间分布准备 cgroup](#)
- [cgroups\(7\)、sysfs\(5\) 手册页](#)

24.4. 通过调整 CPU 权重来控制应用程序的 CPU 时间分布

您需要为 **cpu** 控制器的相关文件分配值，以控制分发到特定 cgroup 树下应用程序的 CPU 时间。

先决条件

- 您有 root 权限。
- 您有要控制 CPU 时间分布的应用程序。
- 您确保相关的应用程序在同一 CPU 上竞争 CPU 时间，如 [为 CPU 时间分布准备 cgroup](#) 中所述。
- 您已挂载了 **cgroups-v2** 文件系统，如 [挂载 cgroups-v2](#) 中所述。
- 您在 **/sys/fs/cgroup/** 根控制组中创建了两级子控制组，如下例所示：

```
...
|— Example
|  |— g1

```



- 您在父控制组中和子控制组中启用了 **cpu** 和 **cpuset** 控制器，如 为 CPU 时间分布准备 cgroup 中所述。

流程

- 配置所需的 CPU 权重以便在控制组群内实现资源限制：

```
# echo "150" > /sys/fs/cgroup/Example/g1/cpu.weight
# echo "100" > /sys/fs/cgroup/Example/g2/cpu.weight
# echo "50" > /sys/fs/cgroup/Example/g3/cpu.weight
```

- 将应用程序的 PID 添加到 **g1**、**g2** 和 **g3** 子组中：

```
# echo "33373" > /sys/fs/cgroup/Example/g1/cgroup.procs
# echo "33374" > /sys/fs/cgroup/Example/g2/cgroup.procs
# echo "33377" > /sys/fs/cgroup/Example/g3/cgroup.procs
```

示例命令可确保所需的应用程序成为 **Example/g*/** 子 cgroup 的成员，并按照这些 cgroup 配置获取发布的 CPU 时间。

已运行进程的子 cgroup (**g1**, **g2**, **g3**) 的权重在父 cgroup (**Example**) 级别上相加。然后，根据对应的权重按比例分布 CPU 资源。

因此，当所有进程同时运行时，内核会根据相应的 cgroup 的 **cpu.weight** 文件为每个进程按比例分配 CPU 时间：

子 cgroup	cpu.weight 文件	CPU 时间分配
g1	150	~50% (150/300)
g2	100	~33% (100/300)
g3	50	~16% (50/300)

cpu.weight 控制器文件的值不是一个百分比。

如果一个进程停止运行，使 cgroup **g2** 没有运行进程，则计算将省略 cgroup **g2**，仅计算 cgroup **g1** 和 **g3** 的帐户权重：

子 cgroup	cpu.weight 文件	CPU 时间分配
g1	150	~75% (150/200)
g3	50	~25% (50/200)



重要

如果子 cgroup 有多个正在运行的进程，则分配给相应 cgroup 的 CPU 时间将平均分配给该 cgroup 的成员进程。

验证

1. 验证应用程序是否运行在指定的控制组中：

```
# cat /proc/33373/cgroup /proc/33374/cgroup /proc/33377/cgroup
0::/Example/g1
0::/Example/g2
0::/Example/g3
```

命令输出显示了运行在 **Example/g*** 子 cgroup 中指定的应用程序的进程。

2. 检查节流应用程序的当前 CPU 消耗：

```
# top
top - 05:17:18 up 1 day, 18:25, 1 user, load average: 3.03, 3.03, 3.00
Tasks: 95 total, 4 running, 91 sleeping, 0 stopped, 0 zombie
%Cpu(s): 18.1 us, 81.6 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.3 hi, 0.0 si, 0.0 st
MiB Mem : 3737.0 total, 3233.7 free, 132.8 used, 370.5 buff/cache
MiB Swap: 4060.0 total, 4060.0 free, 0.0 used. 3373.1 avail Mem

  PID USER   PR NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 33373 root    20  0 18720 1748 1460 R  49.5  0.0 415:05.87 sha1sum
 33374 root    20  0 18720 1756 1464 R  32.9  0.0 412:58.33 sha1sum
 33377 root    20  0 18720 1860 1568 R  16.3  0.0 411:03.12 sha1sum
   760 root    20  0 416620 28540 15296 S  0.3  0.7  0:10.23 tuned
     1 root    20  0 186328 14108 9484 S  0.0  0.4  0:02.00 systemd
     2 root    20  0     0     0     0 S  0.0  0.0  0:00.01 kthread
...
```



注意

为了更清楚地说明，我们强制所有示例进程运行在单个 CPU 上。当在多个 CPU 上使用时，CPU 权重也会应用同样的原则。

注意 **PID 33373**、**PID 33374** 和 **PID 33377** 的 CPU 资源会根据您分配给相应子 cgroup 的权重 150、100、50 进行分配。权重对应于每个应用程序分配的 CPU 时间的大约 50%、33% 和 16%。

其它资源

- [了解控制组群](#)
- [什么是内核资源控制器](#)
- [挂载 cgroups-v2](#)
- [为 CPU 时间分布准备 cgroup](#)
- [资源分发模型](#)
- [cgroups\(7\)、sysfs\(5\) 手册页](#)

第 25 章 使用带有 SYSTEMD 的控制组群版本 1

您可以使用 **systemd** 系统和**服务管理器**及其提供的实用程序来管理 **cgroups**。这也是 **cgroups** 管理的首选方式。

25.1. 控制组群版本 1 中的 SYSTEMD 角色

RHEL 8 通过将 **cgroup** 层次结构与 **systemd** 单元树绑定，将资源管理设置从进程级移到应用程序级。因此，您可以使用 **systemctl** 命令或通过修改 **systemd** 单元文件来管理系统资源。

默认情况下，**systemd** 系统和**服务管理器**利用 **slice**、**scope** 和 **service** 单元来整理和整理控制组中的进程。**systemctl** 命令允许您通过创建自定义 **slices** 来进一步修改此结构。另外，**systemd** 会在 **/sys/fs/cgroup/** 目录中自动挂载重要内核资源控制器的层次结构。

三种 **systemd** 单元类型用于资源控制：

- **Service** - **systemd** 根据单元配置文件启动的进程或一组进程。服务封装指定的进程，以便它们可以作为一个集启动和停止。服务使用以下方法命名：

```
<name>.service
```

- **Scope** - 外部创建的进程组。范围封装通过 **fork()** 函数由任意进程启动和停止的进程，然后在运行时由 **systemd** 注册。例如，用户会话、容器和虚拟机被视为范围。范围命名如下：

```
<name>.scope
```

- **slice** - 一组分级组织单元。片段组织了一个分级，其中放置范围和服务。实际的进程包含在范围或服务中。slice 单元的每个名称对应层次结构中的位置的路径。短划线("-") 字符充当了分隔符的作用，它把路径组件从 **-.slice** root slice 中分隔。在以下示例中：

```
<parent-name>.slice
```

parent-name.slice 是 **parent.slice** 的子分片，它是 **-.slice** root 片段的子分片。**parent-name.slice** 可以有自己的子slice 名为 **parent-name-name2.slice**，以此类推。

service、**scope** 和 **slice** 单元直接映射到控制组层次结构中的对象。激活这些单元后，它们直接映射到从单元名称构建的控制组路径。

以下是控制组群分级的缩写示例：

```
Control group /:
-.slice
├─user.slice
│   └─user-42.slice
│       └─session-c1.scope
│           └─967 gdm-session-worker [pam/gdm-launch-environment]
│               └─1035 /usr/libexec/gdm-x-session gnome-session --autostart
│                   /usr/share/gdm/greeter/autostart
│                       └─1054 /usr/libexec/Xorg vt1 -displayfd 3 -auth /run/user/42/gdm/Xauthority -background none
│                           -noreset -keeppty -verbose 3
│                               └─1212 /usr/libexec/gnome-session-binary --autostart /usr/share/gdm/greeter/autostart
│                                   └─1369 /usr/bin/gnome-shell
│                                       └─1732 ibus-daemon --xim --panel disable
│                                           └─1752 /usr/libexec/ibus-dconf
```

```

| | | | |—1762 /usr/libexec/ibus-x11 --kill-daemon
| | | | |—1912 /usr/libexec/gsd-xsettings
| | | | |—1917 /usr/libexec/gsd-a11y-settings
| | | | |—1920 /usr/libexec/gsd-clipboard
...
├─init.scope
├─┬1 /usr/lib/systemd/systemd --switched-root --system --deserialize 18
└─system.slice
    ├─rngd.service
    │ └─800 /sbin/rngd -f
    ├─systemd-udev.service
    │ └─659 /usr/lib/systemd/systemd-udev
    ├─chronyd.service
    │ └─823 /usr/sbin/chronyd
    ├─auditd.service
    │ └─┬761 /sbin/auditd
    │   └─763 /usr/sbin/sedispatch
    ├─accounts-daemon.service
    │ └─876 /usr/libexec/accounts-daemon
    ├─example.service
    │ └─┬929 /bin/bash /home/jdoe/example.sh
    │   └─4902 sleep 1
    ...

```

上面的例子显示，服务和范围包含进程，并放置在不含自己进程的片段中。

其它资源

- [什么是内核资源控制器](#)
- `systemd.resource-control(5)`, `cgroups(7)`, `fork()`, `fork(2)` manual pages

25.2. 创建临时控制组群

临时 `cgroup` 设置运行时期由单元（服务或范围）消耗的资源限制。

流程

- 要创建一个临时控制组群，使用以下格式的 `systemd-run` 命令：

```
# systemd-run --unit=<name> --slice=<name>.slice <command>
```

此命令会创建并启动临时服务或范围单元，并在此类单元中运行自定义命令。

- `--unit=<name>` 选项为单元取一个名称。如果未指定 `--unit`，则会自动生成名称。
- `--slice=<name>.slice` 选项使您的服务或范围单元成为指定片段的成员。将 `<name>.slice` 替换为现有片段的名称（如 `systemctl -t slice` 输出中所示），或通过传递唯一名称来创建新片段。默认情况下，服务和范围作为 `system.slice` 的成员创建。
- 使用您要在服务或范围单元中输入的命令替换 `<command>`。此时会显示以下信息，以确认您已创建并启动了该服务，或者已成功启动范围：

```
# Running as unit <name>.service
```

- *可选*：在进程完成后保持单元运行，以收集运行时信息：

```
# systemd-run --unit=<name> --slice=<name>.slice --remain-after-exit <command>
```

命令会创建并启动临时服务单元，并在单元中运行自定义命令。`--remain-after-exit` 选项可确保服务在其进程完成后继续运行。

其它资源

- [什么是控制组](#)
- [控制组群中的 systemd 角色](#)
- 在 RHEL 中 [管理 systemd](#)
- [systemd-run \(1\) 手册页](#)

25.3. 创建持久的控制组群

若要为服务分配持久控制组，需要编辑其单元配置文件。配置在系统重启后保留，因此可以用于管理自动启动的服务。

流程

- 要创建一个持久控制组，请输入：

```
# systemctl enable <name>.service
```

以上命令会在 `/usr/lib/systemd/system/` 目录中自动创建一个单元配置文件，并在默认情况下将 `<name>.service` 分配给 `system.slice` 单元。

其它资源

- [了解控制组群](#)
- [控制组群中的 systemd 角色](#)
- 在 RHEL 中 [使用 systemctl 管理系统服务](#)
- [systemd-run\(1\) 手册页](#)

25.4. 在命令行中配置内存资源控制设置

在命令行界面中执行命令是针对进程组设置限制、优先级或控制对硬件资源的访问的一种方式。

流程

- 要限制服务的内存用量，请运行以下命令：

```
# systemctl set-property example.service MemoryMax=1500K
```

命令会立即将 1,500 KB 的内存限值分配给 `example.service` 服务所属的控制组中执行的进程。此配置变体中的 `MemoryMax` 参数在 `/etc/systemd/system.control/example.service.d/50-MemoryMax.conf` 文件中定义，并控制

`/sys/fs/cgroup/memory/system.slice/example.service/memory.limit_in_bytes` 文件的值。

- 另外，要临时限制服务的内存用量，请运行：

```
# systemctl set-property --runtime example.service MemoryMax=1500K
```

命令会立即为 `example.service` 服务分配内存限值。`MemoryMax` 参数被定义，直到 `/run/systemd/system.control/example.service.d/50-MemoryMax.conf` 文件中的下一个重启为止。重启后，整个 `/run/systemd/system.control/` 目录和 `MemoryMax` 会被删除。



注意

`50-MemoryMax.conf` 文件存储内存限制，为 4096 字节的倍数 - 特定于 AMD64 和 Intel 64 的内核页大小。实际的字节数量取决于 CPU 构架。

其它资源

- [什么是控制组](#)
- [什么是内核资源控制器](#)
- [systemd.resource-control\(5\)](#) 和 [cgroups\(7\)](#) manual pages
- [控制组群中的 systemd 角色](#)

25.5. 使用单元文件配置内存资源控制设置

每个持久性单元都由 `systemd` 系统和 `ServiceManager` 监管，并在 `/usr/lib/systemd/system/` 目录中有一个单元配置文件。要更改永久单元的资源控制设置，可在文本编辑器中手动或从命令行界面修改其单元配置文件。

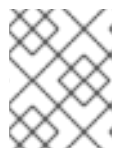
手动修改单元文件是为进程组设置限制、优先级或控制对硬件资源的访问的一种方式。

流程

1. 要限制服务的内存用量，请修改 `/usr/lib/systemd/system/example.service` 文件，如下所示：

```
...
[Service]
MemoryMax=1500K
...
```

以上配置对在控制组中执行的进程的最大内存消耗设定了限制，`example.service` 是该控制组中的一部分。



注意

使用后缀 K、M、G 或 T 将 Kilobyte、Megabyte、Gigabyte 或 Terabyte 识别为一个测量单位。

2. 重新载入所有单元配置文件：

```
# systemctl daemon-reload
```

3. 重启服务：

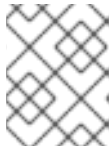
```
# systemctl restart example.service
```

4. 重启系统。

5. (可选) 检查更改是否生效：

```
# cat /sys/fs/cgroup/memory/system.slice/example.service/memory.limit_in_bytes
1536000
```

示例输出显示内存消耗被限制在大约 1,500 KB。

**注意**

memory.limit_in_bytes 文件将内存限制存储为 4096 字节的倍数 - 一个内核页面大小，专用于 AMD64 和 Intel 64。实际的字节数量取决于 CPU 构架。

其它资源

- [了解控制组群](#)
- [内核资源控制器](#)
- [systemd.resource-control\(5\), cgroups\(7\) manual pages](#)
- 在 RHEL 中 [使用 systemctl 管理系统服务](#)
- [控制组群版本 1 中的 systemd 角色](#)

25.6. 删除临时控制组群

如果您不再需要限制、确定或控制对进程组的硬件资源的访问，您可以使用 **systemd** 系统和 **服务管理器** 删除临时控制组 (**cgroup**)。

当服务或范围单元包含的所有进程完成后，临时 **cgroup** 会自动释放。

流程

- 要停止带有所有进程的服务单元，请输入：

```
# systemctl stop <name>.service
```

- 要终止一个或多个单元进程，请输入：

```
# systemctl kill <name>.service --kill-who=PID,... --signal=<signal>
```

命令使用 **--kill-who** 选项从您要终止的控制组中选择进程。要同时终止多个进程，请传递以逗号分隔的 PID 列表。**--signal** 决定要发送到指定进程的 POSIX 信号的类型。默认信号是 **SIGTERM**。

其它资源

- [什么是控制组](#)

- [什么是内核资源控制器](#)
- [systemd.resource-control \(5\) 和 cgroups \(7\) 手册页](#)
- [控制组群版本 1 中的 systemd 角色](#)
- 在 RHEL 中 [管理 systemd](#)

25.7. 删除持久的控制组群

如果您不再需要限制、确定或控制对进程组的硬件资源的访问，您可以使用 **systemd** 系统和服务管理器删除持久性控制组 (**cgroup**)。

当停止或禁用服务或范围单元并删除其配置文件时，永久 **cgroup** 就会发布。

流程

1. 停止服务单元：

```
# systemctl stop <name>.service
```

2. 禁用服务单元：

```
# systemctl disable <name>.service
```

3. 删除相关的单元配置文件：

```
# rm /usr/lib/systemd/system/<name>.service
```

4. 重新载入所有单元配置文件以使更改生效：

```
# systemctl daemon-reload
```

其它资源

- [什么是控制组](#)
- [什么是内核资源控制器](#)
- [systemd.resource-control\(5\), cgroups\(7\), and systemd.kill\(5\) manual pages](#)
- [控制组群中的 systemd 角色](#)
- 在 RHEL 中 [使用 systemctl 管理系统服务](#)

25.8. 列出 SYSTEMD 单元

使用 **systemd** 系统和服务管理器列出其单元。

流程

- 使用 **systemctl** 工具列出系统上所有活动的单元。终端返回一个类似于以下示例的输出：

```

# systemctl
UNIT                                LOAD  ACTIVE SUB    DESCRIPTION
...
init.scope                          loaded active running System and Service Manager
session-2.scope                     loaded active running Session 2 of user jdoe
abrt-ccpp.service                   loaded active exited Install ABRT coredump hook
abrt-oops.service                   loaded active running ABRT kernel log watcher
abrt-vmcore.service                 loaded active exited Harvest vmcores for ABRT
abrt-xorg.service                   loaded active running ABRT Xorg log watcher
...
-.slice                              loaded active active Root Slice
machine.slice                       loaded active active Virtual Machine and Container
Slice system-getty.slice            loaded active active
system-getty.slice
system-lvm2\x2dpvscan.slice         loaded active active system-
lvm2\x2dpvscan.slice
system-sshd\x2dkeygen.slice         loaded active active system-
sshd\x2dkeygen.slice
system-systemd\x2dhibernate\x2dresume.slice loaded active active system-
systemd\x2dhibernate\x2dresume>
system-user\x2druntime\x2ddir.slice loaded active active system-
user\x2druntime\x2ddir.slice
system.slice                        loaded active active System Slice
user-1000.slice                    loaded active active User Slice of UID 1000
user-42.slice                      loaded active active User Slice of UID 42
user.slice                          loaded active active User and Session Slice
...

```

UNIT

还反映控制组层次结构中单元位置的单元名称。与资源控制相关的单元是 *slice*、*scope* 和 *service*

LOAD

指示单元配置文件是否被正确加载。如果单元文件加载失败，该字段包含状态 *error* 而不是 *loaded*。其他单元负载状态为：*stub*、*merge* 和 *masked*。

ACTIVE

高级单元激活状态，其是 **SUB** 的一个泛论。

SUB

低级单元激活状态。可能的值的范围取决于单元类型。

DESCRIPTION

单元内容和功能的描述。

- 列出所有活跃的和不活跃的单元：

```
# systemctl --all
```

- 限制输出中的信息量：

```
# systemctl --type service,masked
```

--type 选项需要一个以逗号分隔的单元类型列表，如 *service* 和 *slice*，或者单元载入状态，如 *loaded* 和 *masked*。

其它资源

- 在 RHEL 中 [使用 systemctl 管理系统服务](#)
- [systemd.resource-control \(5\)](#), [systemd.exec \(5\)](#) 手册页

25.9. 查看 SYSTEMD CGROUPS 层次结构

显示控制组(cgroups)层次结构和在特定 cgroup 中运行的进程。

流程

- 使用 `systemd-cgls` 命令显示系统上整个 cgroups 层次结构。

```
# systemd-cgls
Control group /:
-.slice
  └─user.slice
     └─user-42.slice
        └─session-c1.scope
           └─ 965 gdm-session-worker [pam/gdm-launch-environment]
              └─1040 /usr/libexec/gdm-x-session gnome-session --autostart
                 /usr/share/gdm/greeter/autostart
...
  └─init.scope
     └─ 1 /usr/lib/systemd/systemd --switched-root --system --deserialize 18
        └─system.slice
...
  └─example.service
     └─ 6882 /bin/bash /home/jdoe/example.sh
        └─ 6902 sleep 1
  └─systemd-journald.service
     └─ 629 /usr/lib/systemd/systemd-journald
...

```

示例输出返回整个 cgroups 层次结构，其中最高级别由 slices 组成。

- 使用 `systemd-cgls <resource_controller>` 命令显示由资源控制器过滤的 cgroups 层次结构。

```
# systemd-cgls memory
Controller memory; Control group /:
└─ 1 /usr/lib/systemd/systemd --switched-root --system --deserialize 18
  └─user.slice
     └─user-42.slice
        └─session-c1.scope
           └─ 965 gdm-session-worker [pam/gdm-launch-environment]
...
  └─system.slice
     |
...
  └─chronyd.service
     └─ 844 /usr/sbin/chronyd
  └─example.service

```

```

├─8914 /bin/bash /home/jdoe/example.sh
├─8916 sleep 1
...

```

示例输出列出了与所选控制器交互的服务。

- 使用 **systemctl status <system_unit>** 命令显示特定单元及其 **cgroups** 层次结构部分的详细信息。

```

# systemctl status example.service
● example.service - My example service
   Loaded: loaded (/usr/lib/systemd/system/example.service; enabled; vendor preset:
disabled)
   Active: active (running) since Tue 2019-04-16 12:12:39 CEST; 3s ago
 Main PID: 17737 (bash)
    Tasks: 2 (limit: 11522)
  Memory: 496.0K (limit: 1.5M)
   CGroup: /system.slice/example.service
           └─17737 /bin/bash /home/jdoe/example.sh
             └─17743 sleep 1

Apr 16 12:12:39 redhat systemd[1]: Started My example service.
Apr 16 12:12:39 redhat bash[17737]: The current time is Tue Apr 16 12:12:39 CEST 2019
Apr 16 12:12:40 redhat bash[17737]: The current time is Tue Apr 16 12:12:40 CEST 2019

```

其它资源

- [什么是内核资源控制器](#)
- **systemd.resource-control (5)** 和 **cgroups (7)** 手册页

25.10. 查看资源控制器

查找哪些进程使用哪些资源控制器。

流程

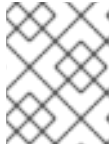
1. 要查看一个进程与哪些资源控制器进行交互，输入 **cat proc/<PID>/cgroup** 命令。

```

# cat /proc/11269/cgroup
12:freezer:/
11:cpuset:/
10:devices:/system.slice
9:memory:/system.slice/example.service
8:pids:/system.slice/example.service
7:hugetlb:/
6:rdma:/
5:perf_event:/
4:cpu,cpuacct:/
3:net_cls,net_prio:/
2:blkio:/
1:name=systemd:/system.slice/example.service

```

输出示例与关注进程相关。在本例中，它是 **PID 11269** 标识的进程，属于 **example.service** 单元。您可以确定该过程是否放置在 **systemd** 单元文件规格定义的正确控制组中。



注意

默认情况下，在资源控制器列表中的项目及其排序对于 **systemd** 启动的所有单元来说是相同的，因为它会自动挂载所有默认资源控制器。

其它资源

- **cgroups (7)** 手册页
- `/usr/share/doc/kernel-doc-<kernel_version>/Documentation/cgroups-v1/` 目录中的文档

25.11. 监控资源消耗

查看当前正在运行控制组(**cgroups**)的列表及其实时资源消耗。

流程

1. 使用 **systemd-cgtop** 命令显示当前正在运行的 **cgroup** 的动态帐户。

```
# systemd-cgtop
Control Group          Tasks %CPU  Memory Input/s Output/s
/                      607  29.8  1.5G   -      -
/system.slice         125   -    428.7M   -      -
/system.slice/ModemManager.service      3   -     8.6M   -      -
/system.slice/NetworkManager.service    3   -    12.8M   -      -
/system.slice/accounts-daemon.service    3   -     1.8M   -      -
/system.slice/boot.mount                 -   -     48.0K   -      -
/system.slice/chronyd.service            1   -     2.0M   -      -
/system.slice/cockpit.socket             -   -     1.3M   -      -
/system.slice/colord.service             3   -     3.5M   -      -
/system.slice/crond.service              1   -     1.8M   -      -
/system.slice/cups.service               1   -     3.1M   -      -
/system.slice/dev-hugepages.mount        -   -    244.0K   -      -
/system.slice/dev-mapper-rhel\x2dswap.swap -   -    912.0K   -      -
/system.slice/dev-mqueue.mount          -   -     48.0K   -      -
/system.slice/example.service            2   -     2.0M   -      -
/system.slice/firewalld.service          2   -    28.8M   -      -
...
```

示例输出显示当前运行的 **cgroups**，按照资源使用量排序（CPU、内存、磁盘 I/O 负载）。这个列表默认每 1 秒刷新一次。因此，它提供了一个动态洞察每个控制组的实际资源使用情况。

其它资源

- **systemd-cgtop (1)** 手册页

第 26 章 使用 CGROUPS-V2 和 SYSTEMD 配置资源管理

systemd 套件的主要功能是服务管理和监督。因此，**systemd** 确保合适的服务在合适的时间启动，并在引导过程中按正确的顺序启动。服务运行时，它们必须顺利运行，才能以最佳的方式使用底层硬件平台。因此，**systemd** 还提供定义资源管理策略和调优各种选项的能力，这可以提高服务的性能。

26.1. 先决条件

- [Linux cgroup](#) 子系统的基础知识。

26.2. 资源分配模型简介

对于资源管理，**systemd** 使用控制组版本 2 (**cgroups-v2**)接口。



注意

默认情况下，RHEL 8 使用 **cgroups-v1**。因此，您必须启用 **cgroups-v2**，以便 **systemd** 可以使用 **cgroups-v2** 接口进行资源管理。有关如何启用 **cgroups-v2** 的更多信息，请参阅 [挂载 cgroups-v2](#)。

要修改系统资源的发布，您可以应用以下一个或多个资源分发模型：

Weights (权重)

通过添加所有子组的权重来分发资源，并根据与所有权重的总和相比的权重为每个子组分配一部分资源。

例如，如果您有 10 个 **cgroups**，每个权重值为 100，则总和为 1000，每个 **cgroup** 接收十分之一的资源。

权重通常用于分发无状态资源。要调整控制组的权重，请使用 `CPUWeight=` 选项。

Limits

cgroup 可以最多可消耗配置的资源量，但您也可以过量使用资源。因此，子组的总限制可能会超过父 **cgroup** 的限制。

要调整控制组的限制，请使用 `MemoryMax=` 选项。

Protections

您可以为 **cgroup** 设置受保护的资源量。如果资源使用率低于保护边界，内核将尽量不惩罚该 **cgroup**，而支持竞争同一资源的其它 **cgroups**。可以过量使用。

要调整控制组的受保护的资源量，请使用 `MemoryLow=` 选项。

Allocations

有限资源特定数量的独占分配，如实时预算。过量使用是可能的。

其它资源

- [使用 systemd 管理 CPU](#)
- [使用 systemd 分配内存资源](#)
- [使用 systemd 配置 I/O 带宽](#)

26.3. 使用 SYSTEMD 分配 CPU 资源

在由 **systemd** 管理的系统上，每个系统服务在其 **cgroup** 中启动。通过启用对 CPU **cgroup** 控制器的支持，系统使用对 CPU 资源的服务感知分布，而不是按进程分布。在服务感知型分发中，每个服务收到的 CPU 时间与系统上运行的所有其他服务的 CPU 时间大致相同，无论组成该服务的进程数量如何。

如果特定的服务需要更多 CPU 资源，您可以通过更改该服务的 CPU 时间分配策略来授予它们。

流程

在使用 **systemd** 时设置 CPU 时间分配策略选项：

1. 在您选择的服务中检查 CPU 时间分配策略选项的分配值：

```
$ systemctl show --property <CPU time allocation policy option> <service name>
```

2. 将 CPU 时间分配策略选项的所需值设置为 root：

```
# systemctl set-property <service name> <CPU time allocation policy option>=<value>
```

cgroup 属性在设置后立即应用。因此，不需要重新启动服务。

验证

- 要验证您是否成功为服务修改了 CPU 时间分配策略选项的所需值，请输入：

```
$ systemctl show --property <CPU time allocation policy option> <service name>
```

其它资源

- [systemd 的 CPU 时间分配策略选项](#)
- [资源分配模型简介](#)

26.4. SYSTEMD 的 CPU 时间分配策略选项

最常用的 CPU 时间分配策略选项包括：

CPUWeight=

在所有其他服务上为特定服务分配**更高的优先级**。您可以从间隔 1 - 10,000 中选择一个值。默认值为 100。

例如，若要将 **httpd.service** 的 CPU 数量与所有其他服务数量相同，可将值设置为 **CPUWeight=200**。

请注意，**CPUWeight=** 仅在可用 CPU 资源已部署时才能应用。

CPUQuota=

为服务分配**绝对 CPU 时间配额**。此选项的值指定服务收到的 CPU 时间相对于可用 CPU 总时间的最大百分比，如 **CPUQuota=30%**。

请注意，**CPUQuota=** 代表特定资源分布模式的限制值，如 **Weights** 和 **Limits**。

有关 **CPUQuota=** 的更多信息，请参阅 **systemd.resource-control(5)** 手册页。

其它资源

- [资源分配模型简介](#)
- [使用 systemd 分配 CPU 资源](#)
- [资源分配模型简介](#)

26.5. 使用 SYSTEMD 分配内存资源

要使用 **systemd** 分配内存资源，请使用任何一个内存配置选项：

- **MemoryMin**
- **MemoryLow**
- **MemoryHigh**
- **MemoryMax**
- **MemorySwapMax**

流程

在使用 **systemd** 时设置内存分配配置选项：

1. 在您选择的服务中检查内存分配配置选项的分配值。

```
$ systemctl show --property <memory allocation configuration option> <service name>
```

2. 将内存分配配置选项所需的值设置为 **root**。

```
# systemctl set-property <service name> <memory allocation configuration option>=<value>
```



注意

cgroup 属性在设置后会被立即应用。因此，不需要重新启动服务。

验证

- 要验证您是否成功为服务更改了内存分配配置选项所需的值，请输入：

```
$ systemctl show --property <memory allocation configuration option> <service name>
```

其它资源

- [systemd 的内存分配配置选项](#)
- [资源分配模型简介](#)

26.6. SYSTEMD 的内存分配配置选项

在使用 **systemd** 配置系统内存分配时，您可以使用以下选项：

MemoryMin

硬内存保护。如果内存使用低于指定的限制，则不会回收 cgroup 内存。

MemoryLow

软内存保护。如果内存使用低于指定的限制，则只有在没有从未保护的 cgroup 中回收内存时才能回收 cgroup 内存。

MemoryHigh

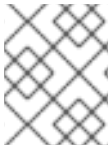
内存节流限制。如果内存使用超过指定的限制，则 cgroup 中的进程会被节流，并置于严重的回收压力下。

MemoryMax

内存用量的绝对限制。您可以使用 kilo(K)、MB(M)、giga(G)、tera(T) 后缀，如 **MemoryMax=1G**。

MemorySwapMax

交换内存使用的硬性限制。



注意

当您用尽内存限值时，内存不足 (OOM) 终止程序将停止正在运行的服务。要防止这种情况，请降低 **OOMScoreAdjust=** 值，以提高内存容错能力。

其它资源

- [使用 systemd 分配内存资源](#)
- [资源分配模型简介](#)

26.7. 使用 SYSTEMD 配置 I/O 带宽

要提高 RHEL 8 中特定服务的性能，您可以使用 **systemd** 为该服务分配 I/O 带宽资源。

要做到这一点，您可以使用以下 I/O 配置选项：

- IOWeight
- IODeviceWeight
- IOReadBandwidthMax
- IOWriteBandwidthMax
- IOReadIOPSMax
- IOWriteIOPSMax

流程

使用 **systemd** 设置 I/O 带宽配置选项：

1. 在您选择的服务中检查 I/O 带宽配置选项的分配值：

```
$ systemctl show --property <I/O bandwidth configuration option> <service name>
```

- 将 I/O 带宽配置选项所需的值设置为 **root** :

```
# systemctl set-property <service name> <I/O bandwidth configuration option>=  
<value>
```

cgroup 属性在设置后会被立即应用。因此，不需要重新启动服务。

验证

- 要验证您是否成功更改了服务的 I/O 带宽配置选项所需的值，请输入：

```
$ systemctl show --property <I/O bandwidth configuration option> <service name>
```

其它资源

- [systemd 的 I/O 带宽配置选项](#)
- [资源分配模型简介](#)

26.8. SYSTEMD 的 I/O 带宽配置选项

要使用 **systemd** 管理块层 I/O 策略，可以使用以下配置选项：

IOWeight

设置默认 I/O 权重。权重值用作计算服务与其他服务相比实际 I/O 带宽数量的基础。

IODeviceWeight

设置特定块设备的 I/O 权重。

例如，**IODeviceWeight=/dev/disk/by-id/dm-name-rhel-root 200**。

IOReadBandwidthMax, IOWriteBandwidthMax

设置每个设备或挂载点的绝对带宽。

例如：**IOWriteBandwidth=/var/log 5M**。



注意

systemd 自动处理文件系统-到-设备转换。

IOReadIOPSMax, IOWriteIOPSMax

每秒输入/输出操作中设置绝对带宽(IOPS)。



注意

只有在块设备使用 CFQ I/O 调度程序时，才支持基于权重的选项。如果设备使用了多队列块 I/O 排队机制，则不支持选项。

其它资源

- [使用 systemd 配置 I/O 带宽](#)
- [资源分配模型简介](#)

26.9. 使用 SYSTEMD 配置 CPuset 控制器

使用 **systemd** 资源管理 API，您可以对服务可以使用的一组 CPU 和 NUMA 节点配置限制。这个限制限制对进程使用的系统资源的访问。请求的配置写在 **cpuset.cpus** 和 **cpuset.mems** 文件中。

但是，可能无法使用请求的配置，因为父 **cgroup** 限制 **cpus** 或 **mems**。若要访问当前的配置，将 **cpuset.cpus.effective** 和 **cpuset.mems.ff** 文件导出到用户。

流程

- 设置 **AllowedCPUs** :

```
# systemctl set-property <service name>.service AllowedCPUs=<value>
```

例如 :

```
# systemctl set-property <service name>.service AllowedCPUs=0-5
```

- 设置 **AllowedMemoryNodes** :

```
# systemctl set-property <service name>.service AllowedMemoryNodes=<value>
```

例如 :

```
# systemctl set-property <service name>.service AllowedMemoryNodes=0
```

第 27 章 使用 SYSTEMD 配置 CPU 关联性和 NUMA 策略

CPU 管理、内存管理和 I/O 带宽选项处理可用资源的分区。

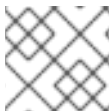
27.1. 使用 SYSTEMD 配置 CPU 关联性

CPU 关联性设置可帮助您将特定进程的访问限制到某些 CPU。实际上，CPU 调度程序永远不会将进程调度到不在进程的关联性掩码中的 CPU 上运行。

默认 CPU 关联性掩码应用到 systemd 管理的所有服务。

要为特定 systemd 服务配置 CPU 关联性掩码，systemd 提供 **CPUAffinity=** 作为单元文件选项和 **/etc/systemd/system.conf** 文件中的管理器配置选项。

CPUAffinity= 单元文件选项 设置 CPU 或 CPU 范围列表，这些范围合并并用作关联性掩码。**/etc/systemd/system.conf** 文件中的 **CPUAffinity** 选项 为进程识别号 (PID)1 和从 PID1 分叉的所有进程定义关联性掩码。然后，您可以基于每个服务覆盖 **CPUAffinity**。



注意

在为特定 systemd 服务配置 CPU 关联性掩码后，您必须重启系统以应用更改。

流程

使用 **CPUAffinity** 单元文件选项为特定 systemd 服务设置 CPU 关联性掩码：

1. 在您选择的服务中检查 **CPUAffinity** 单元文件选项的值：

```
$ systemctl show --property <CPU affinity configuration option> <service name>
```

2. 作为 root 用户，为用作关联性掩码的 CPU 范围设置 **CPUAffinity** 单元文件选项的所需值：

```
# systemctl set-property <service name> CPUAffinity=<value>
```

3. 重新启动服务以应用更改。

```
# systemctl restart <service name>
```

使用 **manager configuration** 选项为特定 systemd 服务设置 CPU 关联性掩码：

1. 编辑 **/etc/systemd/system.conf** 文件：

```
# vi /etc/systemd/system.conf
```

2. 搜索 **CPUAffinity=** 选项并设置 CPU 号
3. 保存编辑后的文件并重新启动服务器以应用更改。

27.2. 使用 SYSTEMD 配置 NUMA 策略

非统一内存访问(NUMA)是一种计算机内存子系统设计，其中内存访问时间取决于相对于处理器的物理内存位置。

靠近 CPU 的内存比不同 CPU 的本地内存(外部内存)或一组 CPU 之间共享的内存具有更低的延迟（本地内存）。

就 Linux 内核而言，NUMA 策略管理内核为进程分配物理内存页面的位置（例如，在哪些 NUMA 节点上）。

systemd 提供单元文件选项 **NUMAPolicy** 和 **NUMAMask**，以控制服务的内存分配策略。

流程

通过 **NUMAPolicy** 单元文件选项设置 NUMA 内存策略：

1. 在您选择的服务中检查 **NUMAPolicy** 单元文件选项的值：

```
$ systemctl show --property <NUMA policy configuration option> <service name>
```

2. 作为根目录，设置 **NUMAPolicy** 单元文件选项所需的策略类型：

```
# systemctl set-property <service name> NUMAPolicy=<value>
```

3. 重新启动服务以应用更改。

```
# systemctl restart <service name>
```

要使用 [Manager] 配置选项设置全局 **NUMAPolicy** 设置：

1. 在 `/etc/systemd/system.conf` 文件中搜索文件的 [Manager] 部分中的 **NUMAPolicy** 选项。
2. 编辑策略类型并保存文件。
3. 重新载入 **systemd** 配置：

```
# systemd daemon-reload
```

4. 重启服务器。



重要

当您配置严格的 NUMA 策略时，例如 **bind**，请确保您也正确地设置了 **CPUAffinity=** 单元文件选项。

其它资源

- [systemd 的 NUMA 策略配置选项](#)
- [systemd.resource-control \(5\)](#), [systemd.exec \(5\)](#) 和 [set_mempolicy \(2\)](#) 手册页。

27.3. SYSTEMD 的 NUMA 策略配置选项

Systemd 提供以下选项来配置 NUMA 策略：

NUMAPolicy

控制已执行进程的 NUMA 内存策略。您可以使用这些策略类型：

- default
- preferred
- bind
- interleave
- local

NUMAMask

控制与所选 NUMA 策略关联的 NUMA 节点列表。
请注意，您不必为以下策略指定 **NUMAMask** 选项：

- default
- local

对于首选策略，列表仅指定单个 NUMA 节点。

其它资源

- [systemd.resource-control \(5\)](#), [systemd.exec \(5\)](#) 和 [set_mempolicy \(2\)](#) 手册页
- [使用 systemd 配置 NUMA](#)

第 28 章 使用 BPF COMPILER COLLECTION 分析系统性能

作为系统管理员，您可以使用 BPF Compiler Collection (BCC) 库创建用于分析 Linux 操作系统性能和收集信息的工具，这些信息可能难以通过其他接口获得。

28.1. 安装 BCC-TOOLS 软件包

安装 **bcc-tools** 软件包，该软件包还会将 BPF Compiler Collection (BCC) 库作为依赖项安装。

流程

1. 安装 **bcc-tools**。

```
# yum install bcc-tools
```

BCC 工具安装在 **/usr/share/bcc/tools/** 目录中。

2. (可选) 检查工具：

```
# ll /usr/share/bcc/tools/  
...  
-rwxr-xr-x. 1 root root 4198 Dec 14 17:53 dcsnoop  
-rwxr-xr-x. 1 root root 3931 Dec 14 17:53 dcstat  
-rwxr-xr-x. 1 root root 20040 Dec 14 17:53 deadlock_detector  
-rw-r--r--. 1 root root 7105 Dec 14 17:53 deadlock_detector.c  
drwxr-xr-x. 3 root root 8192 Mar 11 10:28 doc  
-rwxr-xr-x. 1 root root 7588 Dec 14 17:53 execsnoop  
-rwxr-xr-x. 1 root root 6373 Dec 14 17:53 ext4dist  
-rwxr-xr-x. 1 root root 10401 Dec 14 17:53 ext4slower  
...
```

上表中的 **doc** 目录包含每个工具的文档。

28.2. 使用所选 BCC-TOOLS 进行性能调整

使用 BPF Compiler Collection (BCC) 库中的某些预先创建的程序，以每个事件为基础来高效且安全地分析系统性能。BCC 库中预创建的程序集可作为创建其他程序的示例。

先决条件

- [安装的 bcc-tools 软件包](#)
- 根权限

使用 execsnoop 检查系统进程

1. 在一个终端中运行 **execsnoop** 程序：

```
# /usr/share/bcc/tools/execsnoop
```

2. 在另一个终端中运行，例如：

```
$ ls /usr/share/bcc/tools/doc/
```

以上可创建 **ls** 命令的短时间进程。

3. 运行 **execsnoop** 的终端显示类似如下的输出：

```
PCOMM PID  PPID  RET ARGS
ls 8382 8287  0 /usr/bin/ls --color=auto /usr/share/bcc/tools/doc/
...
```

execsnoop 程序打印出每个占用系统资源的新进程的输出行。它甚至会检测很快运行的程序（如 **ls**）的进程，大多数监控工具也不会进行注册。

execsnoop 输出显示以下字段：

PCOMM

父进程名称。（**ls**）

PID

进程 ID。（**8382**）

PPID

父进程 ID。（**8287**）

RET

exec () 系统调用的返回值(**0**)，其将程序代码加载到新进程中。

ARGS

启动的程序的参数的位置。

要查看 **execsnoop** 的详情、示例和选项，请参阅 `/usr/share/bcc/tools/doc/execsnoop_example.txt` 文件。

有关 **exec()** 的详情，请查看 **exec(3)** 手册页。

使用 **opensnoop** 跟踪命令打开的文件

1. 在一个终端中运行 **opensnoop** 程序：

```
# /usr/share/bcc/tools/opensnoop -n uname
```

以上列出了文件的输出，这些文件仅由 **uname** 命令的进程打开。

2. 在另一个终端中，输入：

```
$ uname
```

以上命令会打开某些在下一步中捕获的文件。

3. 运行 **opensnoop** 的终端显示类似如下的输出：

```
PID  COMM  FD ERR PATH
8596  uname  3  0  /etc/ld.so.cache
8596  uname  3  0  /lib64/libc.so.6
8596  uname  3  0  /usr/lib/locale/locale-archive
...
```

opensnoop 程序在整个系统中监视 **open()** 系统调用，并为 **uname** 尝试打开的每个文件打印一行输出。

opensnoop 输出显示以下字段：

PID

进程 ID。(8596)

COMM

进程名称。(uname)

FD

文件描述符 - **open ()** 返回的值，以指向打开的文件。(3)

ERR

任何错误。

PATH

open () 试图打开的文件的位置。

如果命令尝试读取不存在的文件，则 **FD** 列返回 **-1**，**ERR** 列将打印与相关错误对应的值。因此，**Opennoop** 可以帮助您识别行为不正确的应用程序。

要查看 **opensnoop** 的更多详细信息、示例和选项，请参阅 **/usr/share/bcc/tools/doc/opensnoop_example.txt** 文件。

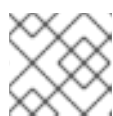
有关 **open()** 的更多信息，请参阅 **open(2)** 手册页。

使用技术检查磁盘上的 I/O 操作

1. 在一个终端中运行 **biotop** 程序：

```
# /usr/share/bcc/tools/biotop 30
```

该命令可让您监控在磁盘中执行 I/O 操作的主要进程。参数确保命令生成 30 秒概述。



注意

如果未提供任何参数，则默认情况下输出屏幕会每 1 秒刷新一次。

2. 在另一个终端输入中，例如：

```
# dd if=/dev/vda of=/dev/zero
```

以上命令从本地硬盘设备读取内容，并将输出写入 **/dev/zero** 文件。此步骤会生成特定的 I/O 流量来演示 **biotop**。

3. 运行 **biotop** 的终端显示类似如下的输出：

```
PID  COMM      D MAJ MIN DISK   I/O Kbytes  AVGms
9568 dd        R 252 0  vda    16294 14440636.0 3.69
48  kswapd0   W 252 0  vda     1763 120696.0 1.65
7571 gnome-shell R 252 0  vda     834 83612.0 0.33
1891 gnome-shell R 252 0  vda    1379 19792.0 0.15
7515 Xorg      R 252 0  vda     280 9940.0 0.28
7579 llvmpipe-1 R 252 0  vda     228 6928.0 0.19
```

```

9515  gnome-control-c  R 252 0  vda      62 6444.0  0.43
8112  gnome-terminal-  R 252 0  vda      67 2572.0  1.54
7807  gnome-software  R 252 0  vda      31 2336.0  0.73
9578  awk              R 252 0  vda      17 2228.0  0.66
7578  llvmpipe-0      R 252 0  vda      156 2204.0  0.07
9581  pgrep           R 252 0  vda      58 1748.0  0.42
7531  InputThread     R 252 0  vda      30 1200.0  0.48
7504  gdbus           R 252 0  vda      3 1164.0  0.30
1983  llvmpipe-1     R 252 0  vda      39 724.0   0.08
1982  llvmpipe-0     R 252 0  vda      36 652.0   0.06
...

```

biotop 输出显示以下字段：

PID

进程 ID。(9568)

COMM

进程名称。(dd)

DISK

执行读操作的磁盘。(vda)

I/O

执行的读操作的数量。(16294)

Kbytes

读操作达到的 Kbytes 量。(14,440,636)

AVGms

读操作的平均 I/O 时间。(3.69)

要查看 **biotop** 的详情、示例和选项，请参阅 `/usr/share/bcc/tools/doc/biotop_example.txt` 文件。

有关 **dd** 的更多信息，请参阅 **dd(1)** 手册页。

使用 **xfsslower** 来公开意料外的慢文件系统操作

1. 在一个终端中运行 **xfsslower** 程序：

```
# /usr/share/bcc/tools/xfsslower 1
```

以上命令测量 XFS 文件系统执行读取、写入、打开或同步 (**fsync**) 操作的时间。**1** 参数可确保程序仅显示比 1 ms 较慢的操作。



注意

如果未提供任何参数，**xfsslower** 默认会显示比 10 ms 慢的操作。

2. 在另一个终端输入中，例如：

```
$ vim text
```

以上命令在 **vim** 编辑器中创建了一个文本文件，用于启动与 XFS 文件系统的某些互动。

3. 运行 **xfsslower** 的终端显示在保存上一步中的文件时：

```

TIME   COMM      PID  T BYTES  OFF_KB  LAT(ms)  FILENAME
13:07:14 b'bash'    4754  R 256   0       7.11 b'vim'
13:07:14 b'vim'     4754  R 832   0       4.03 b'libgpm.so.2.1.0'
13:07:14 b'vim'     4754  R 32    20      1.04 b'libgpm.so.2.1.0'
13:07:14 b'vim'     4754  R 1982  0       2.30 b'vimrc'
13:07:14 b'vim'     4754  R 1393  0       2.52 b'getsriptPlugin.vim'
13:07:45 b'vim'     4754  S 0     0       6.71 b'text'
13:07:45 b'pool'    2588  R 16    0       5.58 b'text'
...

```

上面的每一行代表文件系统中的操作，其用时超过特定阈值。**xfsslower** 非常适合公开可能的文件系统问题，这可能会导致意外的慢速操作。

xfsslower 输出显示以下字段：

COMM

进程名称。(b'bash')

T

操作类型。(R)

- Read
- Write
- Sync

OFF_KB

KB 为单位的文件偏移。(0)

FILENAME

被读、写或同步的文件。

要查看 **xfsslower** 的详情、示例和选项，请参阅 `/usr/share/bcc/tools/doc/xfsslower_example.txt` 文件。

有关 **fsync** 的详情请参考 **fsync(2)** 手册页。

第 29 章 使用内核完整性子系统提高安全性

您可以使用内核完整性子系统的组件来提高系统的保护。了解有关相关组件及其配置的更多信息。

29.1. 内核完整性子系统

完整性子系统是保持系统数据整体完整性的内核的一部分。此子系统有助于保持系统的状态与构建时相同。通过使用此子系统，您可以防止对特定系统文件的不必要的修改。

内核完整性子系统由两个主要组件组成：

完整性测量架构 (IMA)

- 每当执行或通过加密哈希或使用加密密钥签名时，IMA 都会测量文件内容。密钥存储在内核密钥环子系统中。
- IMA 将测量的值放在内核的内存空间中。这可防止系统用户修改测量的值。
- IMA 允许本地和远程方验证测量的值。
- IMA 根据之前存储在内存中测量列表中的值来提供当前文件内容的本地验证。此扩展禁止在当前和之前的测量结果不匹配时对特定文件执行任何操作。

扩展验证模块 (EVM)

- EVM 保护与系统安全性相关的文件的扩展属性（也称为 *xattr*），如 IMA 测量和 SELinux 属性。EVM 使用加密密钥对其相应的值进行加密哈希，或对它们进行签名。密钥存储在内核密钥环子系统中。

内核完整性子系统可以使用受信任的平台模块(TPM)来进一步强化系统安全性。

TPM 是一个带有集成加密密钥的硬件、固件或虚拟组件，它根据用于加密功能的受信任的计算组(TCG)的 TPM 规范而构建。TPM 通常构建为附加到平台的主板的专用硬件。通过提供来自硬件芯片受保护且防篡改区的加密功能，TPM 会免于基于软件的攻击。TPMs 提供以下功能：

- 随机数生成器
- 用于加密密钥的生成器和安全存储
- 哈希生成器
- 远程测试

其它资源

- [安全强化](#)
- [Security-Enhanced Linux \(SELinux\) 的基本和高级配置](#)

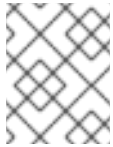
29.2. 可信和加密的密钥

可信密钥 和 *加密密钥* 是增强系统安全性的一个重要部分。

可信和加密的密钥是由使用内核密钥环服务的内核生成的可变长度对称密钥。可以验证密钥的完整性，这意味着可以通过扩展验证模块 (EVM) 来验证并确认正在运行的系统的完整性。用户级别程序只能访问加密的 *Blob* 格式的密钥。

可信密钥

可信密钥需要受信任的平台模块(TPM)芯片，用于创建和加密(密封)密钥。每个 TPM 都有一个主包装密钥，称为存储根密钥，其存储在 TPM 本身。



注意

Red Hat Enterprise Linux 8 支持 TPM 1.2 和 TPM 2.0。如需更多信息，请参阅 [红帽支持受信任的平台模块\(TPM\)吗？](#) 解决方案。

您可以输入以下命令来验证 TPM 2.0 芯片是否已启用：

```
$ cat /sys/class/tpm/tpm0/tpm_version_major
2
```

您还可以启用 TPM 2.0 芯片，并通过机器固件中的设置管理 TPM 2.0 设备。

此外，您还可以使用特定的 TPM 的 *平台配置寄存器* (PCR) 值集密封可信的密钥。PCR 包含一组完整性管理值，它们反映了固件、引导装载程序和操作系统。这意味着 PCR 密封的密钥只能被加密的同一系统上的 TPM 解密。但是，当加载 PCR 密封的可信密钥（添加到密钥环）时，且因此验证其关联的 PCR 值时，可以使用新的（或未来的）PCR 值对其进行更新，以便可以引导新的内核。您可以将单个密钥另存为多个 blob，每个密钥都有不同的 PCR 值。

加密的密钥

加密的密钥不需要 TPM，因为它们使用内核高级加密标准(AES)，这使其比可信密钥更快。加密的密钥是使用内核生成的随机数字创建的，并在导入到用户空间 Blob 时由 *主密钥* 加密。

主密钥可以是可信密钥或用户密钥。如果主密钥不被信任，加密的密钥的安全性仅与用于加密它的用户密钥一样安全。

29.3. 使用可信密钥

您可以使用 `keyctl` 工具创建、导出、加载和更新可信密钥来提高系统安全性。

先决条件

- 对于 64 位 ARM 架构和 IBM Z，可信内核模块已载入。

```
# modprobe trusted
```

有关如何载入内核模块的更多信息，请参阅 [在系统运行时载入内核模块](#)。

- 受信任的平台模块(TPM)已启用并处于活动状态。请参阅 [内核完整性子系统](#) 以及 [受信任的和加密的密钥](#)。



注意

Red Hat Enterprise Linux 8 支持 TPM 1.2 和 TPM 2.0。如果使用 TPM 1.2，请跳过第 1 步。

流程

1. 使用具有持久句柄的 SHA-256 主存储密钥创建一个 2048 位 RSA 密钥，例如 `81000001`，使用以下工具之一：

- a. 使用 **tss2** 软件包：

```
# TPM_DEVICE=/dev/tpm0 tsscreateprimary -hi o -st
Handle 80000000
# TPM_DEVICE=/dev/tpm0 tssevictcontrol -hi o -ho 80000000 -hp 81000001
```

- b. 使用 **tpm2-tools** 软件包：

```
# tpm2_createprimary --key-algorithm=rsa2048 --key-context=key.ctxt
name-alg:
  value: sha256
  raw: 0xb
...
sym-keybits: 128
rsa: xxxxxx...

# tpm2_evictcontrol -c key.ctxt 0x81000001
persistentHandle: 0x81000001
action: persisted
```

2. 创建可信密钥：

- a. 通过使用 TPM 2.0 和 **keyctl add trusted <NAME> "new <KEY_LENGTH> keyhandle=<PERSISTENT-HANDLE> [options]" <KEYRING>** 语法。在本例中，持久性句柄为 `81000001`。

```
# keyctl add trusted kmk "new 32 keyhandle=0x81000001" @u
642500861
```

命令创建一个名为 **kmk** 的可信密钥，长度为 **32** 字节（256 位），并将其放置在用户密钥环 (**@u**) 中。密钥长度为 32 到 128 字节（256 到 1024 位）。

- b. 通过使用 TPM 1.2 和 **keyctl add trusted <NAME> "new <KEY_LENGTH>" <KEYRING>** 语法：

```
# keyctl add trusted kmk "new 32" @u
```

3. 列出内核密钥环的当前结构：

```
# keyctl show
Session Keyring
  -3 --alswrv 500 500 keyring: ses 97833714 --alswrv 500 -1 \ keyring: uid.1000
642500861 --alswrv 500 500 \ trusted: kmk
```

4. 使用可信密钥的序列号将密钥导出到用户空间 blob 中：

```
# keyctl pipe 642500861 > kmk.blob
```

命令使用 **pipe** 子命令和 **kmk** 的序列号。

5. 从 user-space blob 加载可信密钥：

-


```
# keyctl add trusted kmk "load `cat kmk.blob`" @u
268728824
```

- 创建使用 TPM 密封的可信密钥的安全加密密钥(kmk)。按照此语法：`keyctl add encrypted <NAME> "new [FORMAT] <KEY_TYPE>:<PRIMARY_KEY_NAME> <KEY_LENGTH>" <KEYRING>`:

```
# keyctl add encrypted encr-key "new trusted:kmk 32" @u
159771175
```

其它资源

- [keyctl \(1\) 手册页](#)
- [可信和加密的密钥](#)
- [内核密钥保留服务](#)
- [内核完整性子系统](#)

29.4. 使用加密密钥

您可以通过管理加密密钥来改进不提供受信任的平台模块(TPM)的系统上的系统安全性。

先决条件

- 对于 64 位 ARM 架构和 IBM Z，载入 **encrypted-keys** 内核模块：

```
# modprobe encrypted-keys
```

有关如何载入内核模块的更多信息，请参阅 [在系统运行时载入内核模块](#)。

流程

- 使用随机数字序列生成用户密钥。

```
# keyctl add user kmk-user "$(dd if=/dev/urandom bs=1 count=32 2>/dev/null)" @u
427069434
```

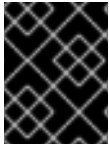
命令生成名为 **kmk-user** 的用户密钥，该密钥充当 *主密钥*，用于密封实际加密的密钥。

- 使用上一步中的主密钥生成加密密钥：

```
# keyctl add encrypted encr-key "new user:kmk-user 32" @u
1012412758
```

- 另外，还可列出指定用户密钥环中的所有密钥：

```
# keyctl list @u
2 keys in keyring:
427069434: --alswrv 1000 1000 user: kmk-user
1012412758: --alswrv 1000 1000 encrypted: encr-key
```



重要

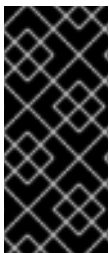
未由可信主密钥密封的加密密钥仅作为用于加密它们的用户主密钥（随机数字密钥）安全。因此，尽可能安全地加载主用户密钥，最好在引导过程的早期加载。

其它资源

- [keyctl \(1\) 手册页](#)
- [内核密钥保留服务](#)

29.5. 启用 IMA 和 EVM

您可以启用并配置完整性测量架构(IMA)和扩展验证模块(EVM)以提高操作系统的安全性。



重要

始终与 IMA 一起启用 EVM。

虽然您可以单独启用 EVM，但 EVM appraisal 仅由 IMA appraisal 规则触发。因此，EVM 不会阻止 SELinux 属性等文件元数据。如果文件元数据被离线篡改，EVM 只能防止文件元数据更改。它不会阻止文件访问，如执行文件。

先决条件

- 安全引导被临时禁用。



注意

启用安全引导后，`ima_appraise=fix` 内核命令行参数无法正常工作。

- **securityfs** 文件系统挂载到 `/sys/kernel/security/` 目录，并且 `/sys/kernel/security/integrity/ima/` 目录存在。您可以使用 `mount` 命令验证 **securityfs** 挂载的位置：

```
# mount
...
securityfs on /sys/kernel/security type securityfs (rw,nosuid,nodev,noexec,relatime)
...
```

- **systemd** 服务管理器打了补丁，以在引导时支持 IMA 和 EVM。使用以下命令验证：

```
# grep <options> pattern <files>
```

例如：

```
# dmesg | grep -i -e EVM -e IMA -w
[ 0.598533] ima: No TPM chip found, activating TPM-bypass!
[ 0.599435] ima: Allocated hash algorithm: sha256
[ 0.600266] ima: No architecture policies found
[ 0.600813] evm: Initialising EVM extended attributes:
[ 0.601581] evm: security.selinux
[ 0.601963] evm: security.ima
```

```
[ 0.602353] evm: security.capability
[ 0.602713] evm: HMAC attrs: 0x1
[ 1.455657] systemd[1]: systemd 239 (239-74.el8_8) running in system mode. (+PAM
+AUDIT +SELINUX +IMA -APPARMOR +SMACK +SYSVINIT +UTMP +LIBCRYPTSETUP
+GCRYPT +GNUTLS +ACL +XZ +LZ4 +SECCOMP +BLKID +ELFUTILS +KMOD +IDN2 -
IDN +PCRE2 default-hierarchy=legacy)
[ 2.532639] systemd[1]: systemd 239 (239-74.el8_8) running in system mode. (+PAM
+AUDIT +SELINUX +IMA -APPARMOR +SMACK +SYSVINIT +UTMP +LIBCRYPTSETUP
+GCRYPT +GNUTLS +ACL +XZ +LZ4 +SECCOMP +BLKID +ELFUTILS +KMOD +IDN2 -
IDN +PCRE2 default-hierarchy=legacy)
```

流程

1. 在 *修复* 模式下为当前引导条目启用 IMA 和 EVM，并通过添加以下内核命令行参数来允许用户收集和更新 IMA 测量：

```
# grubby --update-kernel=/boot/vmlinuz-$(uname -r) --
args="ima_policy=appraise_tcb ima_appraise=fix evm=fix"
```

该命令在 *fix* 模式下为当前引导条目启用 IMA 和 EVM，并允许用户收集和更新 IMA 测量。

ima_policy=appraise_tcb 内核命令行参数确保内核使用默认的可信计算基础(TCB)测量策略和评估步骤。评估步骤禁止访问之前和当前测量结果不匹配的文件。

2. 重启以使更改生效。
3. 可选：验证参数是否已添加到内核命令行中：

```
# cat /proc/cmdline
BOOT_IMAGE=(hd0,msdos1)/vmlinuz-4.18.0-167.el8.x86_64 root=/dev/mapper/rhel-root
ro crashkernel=auto resume=/dev/mapper/rhel-swap rd.lvm.lv=rhel/root
rd.lvm.lv=rhel/swap rhgb quiet ima_policy=appraise_tcb ima_appraise=fix evm=fix
```

4. 创建一个内核主密钥来保护 EVM 密钥：

```
# keyctl add user kmk "$(dd if=/dev/urandom bs=1 count=32 2> /dev/null)" @u
748544121
```

kmk 完全保留在内核空间内存中。**kmk** 的 32 字节长值是从 **/dev/urandom** 文件中的随机字节生成的，并放在用户(@u)密钥环中。密钥序列号位于前面输出的第一行。

5. 根据 **kmk** 创建加密的 EVM 密钥：

```
# keyctl add encrypted evm-key "new user:kmk 64" @u
641780271
```

命令使用 **kmk** 生成并加密 64 字节长用户密钥（名为 **evm-key**），并将其放在用户(@u)密钥环中。密钥序列号位于前面输出的第一行。



重要

用户密钥必须命名为 **evm-key**，因为它是 EVM 子系统预期使用的且正在使用的名称。

6. 为导出的密钥创建一个目录。

```
# mkdir -p /etc/keys/
```

7. 搜索 **kmk**，并将其未加密的值导出到新目录中。

```
# keyctl pipe $(keyctl search @u user kmk) > /etc/keys/kmk
```

8. 搜索 **evm-key**，并将其加密值导出到新目录中。

```
# keyctl pipe $(keyctl search @u encrypted evm-key) > /etc/keys/evm-key
```

evm-key 已在早期由内核主密钥加密。

9. 可选：查看新创建的密钥。

```
# keyctl show
Session Keyring
974575405 --alswrv 0 0 keyring: ses 299489774 --alswrv 0 65534 | keyring:
uid.0 748544121 --alswrv 0 0 | user: kmk
641780271 --alswrv 0 0 \_ encrypted: evm-key

# ls -l /etc/keys/
total 8
-rw-r--r--. 1 root root 246 Jun 24 12:44 evm-key
-rw-r--r--. 1 root root 32 Jun 24 12:43 kmk
```

10. 可选：如果密钥已从密钥环中删除，例如在系统重启后，您可以导入已导出的 **kmk** 和 **evm-key**，而不是创建新密钥。

- a. 导入 **kmk**。

```
# keyctl add user kmk "$(cat /etc/keys/kmk)" @u
451342217
```

- b. 导入 **evm-key**。

```
# keyctl add encrypted evm-key "load $(cat /etc/keys/evm-key)" @u
924537557
```

11. 激活 EVM。

```
# echo 1 > /sys/kernel/security/evm
```

12. 重新标记整个系统。

```
# find / -fstype xfs -type f -uid 0 -exec head -n 1 '{}' >/dev/null \;
```



警告

在不重新标记系统的情况下启用 IMA 和 EVM 可能会导致系统上的大多数文件无法访问。

验证

- 验证 EVM 已初始化。

```
# dmesg | tail -1
[...] evm: key initialized
```

其它资源

- [grep \(1\)手册页](#)
- [内核完整性子系统](#)
- [可信和加密的密钥](#)

29.6. 使用完整性测量架构收集文件哈希

在 *测量* 阶段，您可以创建文件哈希，并将其存储为这些文件的扩展属性(*xattrs*)。通过文件哈希，您可以生成基于 RSA 的数字签名或基于 Hash 的消息身份验证代码(HMAC-SHA1)，从而防止对扩展属性的离线篡改攻击。

先决条件

- IMA 和 EVM 已启用。如需更多信息，请参阅 [启用完整性测量架构和扩展验证模块](#)。
- 有效的可信密钥或加密的密钥保存在内核密钥环中。
- **ima-evm-utils**、**attr** 和 **keyutils** 软件包已安装。

流程

1. 创建测试文件：

```
# echo <Test_text> > test_file
```

IMA 和 EVM 确保 **test_file** 示例文件已分配了哈希值，该值被存储为其扩展属性。

2. 检查文件的扩展属性：

```
# getfattr -m . -d test_file
# file: test_file
security.evm=0sAnDly4VPA0HArpPO/EquitnNyBql
security.ima=0sAQOEDeuUnWzwwKYk+n66h/vby3eD
```

示例输出显示带有 IMA 和 EVM 哈希值和 SELinux 上下文的扩展属性。EVM 添加了一个与其他属性相关的 **security.evm** 扩展属性。此时，您可以在 **security.evm** 上使用 **evmctl** 工具来生成基于 RSA 的数字签名或基于哈希的消息身份验证代码(HMAC-SHA1)的数字签名。

其它资源

- [安全强化](#)

第 30 章 使用 RHEL 系统角色永久配置内核参数

您可以使用 **kernel_settings** RHEL 系统角色一次性在多个客户端上配置内核参数。这个解决方案：

- 提供带有有效输入设置的友好接口。
- 保留所有预期的内核参数。

从控制计算机运行 **kernel_settings** 角色后，内核参数将立即应用于受管系统，并在重新启动后保留。



重要

请注意，通过 RHEL 渠道提供的 RHEL 系统角色作为默认的 AppStream 存储库中的 RPM 软件包提供给 RHEL 客户。RHEL 系统角色也可以作为一个集合提供给具有通过 Ansible Automation Hub 的 Ansible 订阅的客户。

30.1. KERNEL_SETTINGS RHEL 系统角色简介

RHEL 系统角色是一组角色，其提供一致的配置接口来远程管理多个系统。

RHEL 系统角色是为了使用 **kernel_settings** RHEL 系统角色自动化内核的配置而引入的。**rhel-system-roles** 软件包包含这个系统角色以及参考文档。

要将内核参数以自动化方式应用到一个或多个系统，请在 **playbook** 中使用 **kernel_settings** 角色和您选择的一个或多个角色变量。**playbook** 是一个或多个人类可读的 **play** 的列表，采用 **YAML** 格式编写。

您可以使用清单文件来定义您希望 Ansible 根据 **playbook** 来配置的一组系统。

使用 **kernel_settings** 角色，您可以配置：

- 使用 **kernel_settings_sysctl** 角色变量的内核参数
- 使用 **kernel_settings_sysfs** 角色变量的各种内核子系统、硬件设备和设备驱动程序
- **systemd** 服务管理器的 CPU 相关性，并使用 **kernel_settings_systemd_cpu_affinity** 角色变量处理其分叉
- 内核内存子系统使用 **kernel_settings_transparent_hugepages** 和 **kernel_settings_transparent_hugepages_defrag** 角色变量透明巨页

其它资源

- [/usr/share/ansible/roles/rhel-system-roles.kernel_settings/README.md](#) 文件
- [/usr/share/doc/rhel-system-roles/kernel_settings/](#) directory
- [使用 playbook](#)
- [如何构建清单](#)

30.2. 使用 KERNEL_SETTINGS RHEL 系统角色应用所选内核参数

按照以下步骤准备并应用 Ansible **playbook** 来远程配置内核参数，从而对多个受管操作系统产生持久性。

先决条件

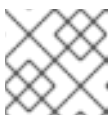
- 您已准备好控制节点和受管节点
- 以可在受管主机上运行 playbook 的用户登录到控制节点。
- 用于连接到受管节点的帐户具有 **sudo** 权限。

流程

1. 创建一个包含以下内容的 playbook 文件，如 **~/playbook.yml**：

```
---
- name: Configure kernel settings
  hosts: managed-node-01.example.com
  roles:
    - rhel-system-roles.kernel_settings
  vars:
    kernel_settings_sysctl:
      - name: fs.file-max
        value: 400000
      - name: kernel.threads-max
        value: 65536
    kernel_settings_sysfs:
      - name: /sys/class/net/lo/mtu
        value: 65000
    kernel_settings_transparent_hugepages: madvise
```

- **name**：可选键，其将一个任意字符串与 play 关联来作为标签的，并确定该 play 的用途。
- **hosts**：play 中的键，其指定运行 play 的主机。此键的值或值可以作为被管理的主机的单独名称提供，也可以作为 **inventory** 文件中定义的一组主机提供。
- **vars**：playbook 的一部分，表示一个变量的列表，其中包含所选的内核参数名称以及它们必须设置成的值。
- **role**：指定哪个 RHEL 系统角色将配置 **vars** 部分中提到的参数和值的键。



注意

您可以修改 playbook 中的内核参数及其值以符合您的需要。

2. 验证 playbook 语法：

```
$ ansible-playbook --syntax-check ~/playbook.yml
```

请注意，这个命令只验证语法，不会防止错误但有效的配置。

3. 运行 playbook：

```
$ ansible-playbook ~/playbook.yml
```

4. 重启您的受管主机并检查受影响的内核参数，以验证是否应用了更改并在重启后保留。

其它资源

- [/usr/share/ansible/roles/rhel-system-roles.kernel_settings/README.md](#) 文件
- [/usr/share/doc/rhel-system-roles/kernel_settings/](#) directory
- [使用 Playbook](#)
- [使用变量](#)
- [角色](#)

第 31 章 使用高级错误报告

当您使用 **高级错误报告 (AER)** 时，您会收到 **Peripheral Component Interconnect Express (PCIe)** 设备的错误事件通知。RHEL 默认启用此内核功能，并收集内核日志中报告的错误。此外，如果您使用 **rasdaemon** 程序，这些错误会被解析并存储在其数据库中。

31.1. AER 概述

高级错误报告 (AER) 是一个内核功能，可为 **Peripheral Component Interconnect Express (PCIe)** 设备提供增强的错误报告。**AER** 内核驱动程序附加在支持 **PCIe AER** 功能的 root 端口，以便：

- 收集全面的错误信息
- 向用户报告错误
- 执行错误恢复操作

当 **AER** 捕获到错误时，它会将向控制台发送一条 **错误** 消息。对于可修复的错误，控制台输出是一条 **警告**。

例 31.1. AER 输出示例

```
Feb 5 15:41:33 hostname kernel: pcieport 10003:00:00.0: AER: Corrected error received:
id=ae00
Feb 5 15:41:33 hostname kernel: pcieport 10003:00:00.0: AER: Multiple Corrected error received:
id=ae00
Feb 5 15:41:33 hostname kernel: pcieport 10003:00:00.0: PCIe Bus Error: severity=Corrected,
type=Data Link Layer, id=0000(Receiver ID)
Feb 5 15:41:33 hostname kernel: pcieport 10003:00:00.0: device [8086:2030] error
status/mask=000000c0/00002000
Feb 5 15:41:33 hostname kernel: pcieport 10003:00:00.0: [ 6] Bad TLP
Feb 5 15:41:33 hostname kernel: pcieport 10003:00:00.0: [ 7] Bad DLLP
Feb 5 15:41:33 hostname kernel: pcieport 10003:00:00.0: AER: Multiple Corrected error received:
id=ae00
Feb 5 15:41:33 hostname kernel: pcieport 10003:00:00.0: PCIe Bus Error: severity=Corrected,
type=Data Link Layer, id=0000(Receiver ID)
Feb 5 15:41:33 hostname kernel: pcieport 10003:00:00.0: device [8086:2030] error
status/mask=00000040/00002000
```

31.2. 收集并显示 AER 消息

要收集和显示 AER 消息，请使用 **rasdaemon** 程序。

流程

1. 安装 **rasdaemon** 软件包。

```
# yum install rasdaemon
```

2. 启用并启动 **rasdaemon** 服务。

```
# systemctl enable --now rasdaemon
```

```
Created symlink /etc/systemd/system/multi-user.target.wants/rasdaemon.service →  
/usr/lib/systemd/system/rasdaemon.service.
```

3. 发出 **ras-mc-ctl** 命令。

```
# ras-mc-ctl --summary
```

```
# ras-mc-ctl --errors
```

命令显示日志记录错误的摘要（**--summary** 选项），或者显示存储在错误数据库中的错误（**--errors** 选项）。

其它资源

- [rasdaemon\(8\) 手册页](#)
- [ras-mc-ctl\(8\) 手册页](#)