Red Hat Enterprise Linux 7
资源管理指南

管理 Red Hat Enterprise Linux 7 的系统资源

作者：Peter Ondrejka 作者：Douglas Silas 作者：Martin Prpič
作者：Rüdiger Landmann
翻译、校对：李雪丹 – Xuedan (Evangeline) Li
校对、编辑：傅同杰 – Tongjie (Tony) Fu
校对、责任编辑：鄭中 – Chester Cheng

Red Hat Enterprise Linux 7 资源管理指南

管理 Red Hat Enterprise Linux 7 的系统资源

作者：Peter Ondrejka
Red Hat 出版中心
pondrejk@redhat.com

作者：Douglas Silas
Red Hat 出版中心
dhensley@redhat.com

作者：Martin Prpič
Red Hat 产品安全部
mprpic@redhat.com

作者：Rüdiger Landmann
Red Hat 出版中心
r.landmann@redhat.com

翻译、校对：李雪丹 – Xuedan (Evangeline) Li
澳大利亚昆士兰大学・笔译暨口译研究生院
527343530@qq.com

校对、编辑：傅同杰 – Tongjie (Tony) Fu
红帽公司・全球服务部
tfu@redhat.com

校对、责任编辑：鄭中 – Chester Cheng
红帽公司・全球服务部 & 澳大利亚昆士兰大学・笔译暨口译研究生院
ccheng@redhat.com, chester.cheng@uq.edu.au

法律通告

Copyright © 2015 Red Hat, Inc.

This document is licensed by Red Hat under the Creative Commons Attribution-ShareAlike 3.0 Unported License. If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

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, 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 Software Collections 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

对 Red Hat Enterprise Linux 7 的系统资源进行管理。

目录

第 1 章 控制群组简介 **2**
1.1. 什么是控制群组 2
1.2. cgroup 的默认层级 2
1.3. Linux Kernel 的资源管控器 4
1.4. 附加资源 4

第 2 章 使用控制群组 **7**
2.1. 创建控制群组 7
2.2. 删除控制群组 8
2.3. 修改 cgroup 8
2.4. 获得关于控制群组的信息 12
2.5. 附加资源 15

第 3 章 使用 libcgroup 工具 **17**
3.1. 挂载层级 17
3.2. 卸载层级 19
3.3. 创建控制群组 19
3.4. 删除控制群组 20
3.5. 设定 cgroup 参数 20
3.6. 将进程移至控制群组 22
3.7. 启动控制群组的进程 23
3.8. 获区关于控制群组的信息 23
3.9. 附加资源 24

第 4 章 控制群组应用示例 **26**
4.1. 定义数据库 I/O 的优先级 26
4.2. 定义网络流量的优先级 27

附录 A. 子系统和可调参数 **29**
A.1. blkio 29
A.2. cpu 34
A.3. cpuacct 36
A.4. cpuset 37
A.5. devices 39
A.6. freezer 40
A.7. memory 40
A.8. net_cls 46
A.9. net_prio 46
A.10. ns 47
A.11. perf_event 47
A.12. 常用可调参数 47
A.13. 附加资源 49

附录 B. 修订历史 **51**

第 1 章 控制群组简介

1.1. 什么是控制群组

控制群组（control group）（在此指南中简写为 *cgroup*）是 Linux kernel 的一项功能：在一个系统中运行的层级制进程组，您可对其进行资源分配（如 CPU 时间、系统内存、网络带宽或者这些资源的组合）。通过使用 cgroup，系统管理员在分配、排序、拒绝、管理和监控系统资源等方面，可以进行精细化控制。硬件资源可以在应用程序和用户间智能分配，从而增加整体效率。

控制群组可对进程进行层级式分组并标记，并对其可用资源进行限制。传统情况下，所有的进程分得的系统资源数量相近，管理员用进程 *niceness* 值进行调节。而用此方法，包含大量进程的应用程序可以比包含少量进程的应用程序获得更多资源，这与应用程序的重要程度无关。

通过将 cgroup 层级系统与 systemd 单位树捆绑，Red Hat Enterprise Linux 7 可以把资源管理设置从进程级别移至应用程序级别。因此，您可以使用 **systemctl** 指令，或者通过修改 systemd 单位文件来管理系统资源。详情请参阅〈第 2 章 *使用控制群组*〉。

在 Red Hat Enterprise Linux 之前的版本中，系统管理员使用 *libcgroup* 软件包中的 **cgconfig** 指令来建立自定义 cgroup 层级。但现在，这个软件包已经过时也不被推荐使用，因为它很容易与默认的 cgroup 层级产生冲突。然而，在一些特定情况下，*libcgroup* 仍然可用，如 **systemd** 不可用时，或使用 *net-prio* 子系统时。详情请参考〈第 3 章 *使用 libcgroup 工具*〉。

上述工具提供了高阶接口，用与 Linux kernel 中的 cgroup 管控器（也称为子系统）互动。用于资源管理的主要 cgroup 管控器是 *cpu*、*memory* 和 *blkio*。如需参考默认情况下被激活的管控器列表，请参阅〈Red Hat Enterprise Linux 7 中可用的管控器〉；如需参考资源管控器详述及可配置参数，请参阅〈kernel 管控器专项介绍〉。

1.2. cgroup 的默认层级

默认情况下，**systemd** 会自动创建 *slice*、*scope* 和 *service* 单位的层级，来为 cgroup 树提供统一结构。使用 **systemctl** 指令，您可以通过创建自定义 slice 进一步修改此结构，详情请参阅〈第 2.1 节 "创建控制群组"〉。**systemd** 也自动为 **/sys/fs/cgroup/** 目录中重要的 kernel 资源管控器（参见〈Red Hat Enterprise Linux 7 中可用的管控器〉）挂载层级。

警告

虽然不推荐使用 **libcgroup** 软件包中的 **cgconfig** 工具，但它可以为 **systemd** （尤其是 **net-prio** 管控器）暂不支持的管控器挂载、处理层级。永远不要使用 **libcgropup** 工具去修改 **systemd** 默认挂载的层级，否则可能会导致异常情况。在 Red Hat Enterprise Linux 后续版本中，**libcgroup** 软件库将会被移除。更多关于如何使用 **cgconfig** 的信息，请参考〈第 3 章 *使用 libcgroup 工具*〉。

systemd 的单位类型

系统中运行的所有进程，都是 **systemd** init 进程的子进程。在资源管控方面，systemd 提供了三种单位类型（如需 **systemd** 单位类型完整列表，请参阅《Red Hat Enterprise Linux 7 系统管理员指南 · *使用 systemd 管理 service*》）：

- **service** —— 一个或一组进程，由 **systemd** 依据单位配置文件启动。service 对指定进程进行封装，这样进程可以作为一个整体被启动或终止。service 参照以下方式命名：

```
name.service
```

其中，*name* 代表服务名称。

- **scope** —— 一组外部创建的进程。由强制进程通过 **fork()** 函数启动和终止、之后被 **systemd** 在运行时注册的进程，scope 会将其封装。例如：用户会话、 容器和虚拟机被认为是 scope。scope 的命名方式如下：

```
name.scope
```

其中，*name* 代表 scope 名称。

- **slice** —— 一组按层级排列的单位。slice 并不包含进程，但会组建一个层级，并将 scope 和 service 都放置其中。真正的进程包含在 scope 或 service 中。在这一被划分层级的树中，每一个 slice 单位的名字对应通向层级中一个位置的路径。小横线（"**-**"）起分离路径组件的作用。例如，如果一个 slice 的名字是：

```
parent-name.slice
```

这说明 *parent-name*.**slice** 是 *parent*.**slice** 的一个子 slice。这一子 slice 可以再拥有自己的子 slice，被命名为：*parent-name-name2*.**slice**，以此类推。

根 slice 的表示方式：

```
-.slice
```

service、scope 和 slice 单位直接映射到 cgroup 树中的对象。当这些单位被激活，它们会直接一一映射到由单位名建立的 cgroup 路径中。例如，*ex.service* 属于 *test-waldo.slice*，会直接映射到 cgroup **test.slice/test-waldo.slice/ex.service/** 中。

service、scope 和 slice 是由系统管理员手动创建或者由程序动态创建的。默认情况下， 操作系统会定义一些运行系统必要的内置 service。另外，默认情况下，系统会创建四种 slice：

- **-.slice** —— 根 slice；
- **system.slice** —— 所有系统 service 的默认位置；
- **user.slice** —— 所有用户会话的默认位置；
- **machine.slice** —— 所有虚拟机和 Linux 容器的默认位置。

请注意，所有的用户会话、虚拟机和容器进程会被自动放置在一个单独的 scope 单元中。而且，所有的用户会分得一个隐含子 slice（implicit subslice）。除了上述的默认配置，系统管理员可能会定义新的 slice，并将 service 和 scope 置于其中。

以下是一个 cgroup 树的简化例子。使用〈第 2.4 节 "获得关于控制群组的信息"〉中记述的 **systemd-cgls** 指令，这一输出就会出现：

```
├─1 /usr/lib/systemd/systemd --switched-root --system --deserialize 20
├─user.slice
│ └─user-1000.slice
│   └─session-1.scope
│     ├─11459 gdm-session-worker [pam/gdm-password]
│     ├─11471 gnome-session --session gnome-classic
│     ├─11479 dbus-launch --sh-syntax --exit-with-session
│     ├─11480 /bin/dbus-daemon --fork --print-pid 4 --print-address 6 --
session
│     ...
│
└─system.slice
```

```
  ├─systemd-journald.service
  │ └─422 /usr/lib/systemd/systemd-journald
  ├─bluetooth.service
  │ └─11691 /usr/sbin/bluetoothd -n
  ├─systemd-localed.service
  │ └─5328 /usr/lib/systemd/systemd-localed
  ├─colord.service
  │ └─5001 /usr/libexec/colord
  ├─sshd.service
  │ └─1191 /usr/sbin/sshd -D
  │
  ...
```

如您所见，service 和 scope 包含进程，但被放置在不包含它们自身进程的 slice 里。唯一例外是位于特殊 systemd.slice 中的 PID 1。请注意，**-.slice** 未被显示，因为它被整体树的根隐性识别。

service 和 slice 单位可通过〈第 2.3.2 节 “修改单位文件”〉中的永久单位文件来配置；或者对 PID 1 进行 API 调用（如需要 API 的详细信息，请参阅〈“在线文档 ”一节〉），在运行时动态创建。scope 单位只能以第一种方式创建。API 调用动态创建的单位是临时的，并且仅在运行时存在。一旦结束、被关闭或者系统重启，临时单位会被自动释放。

1.3. Linux Kernel 的资源管控器

资源管控器（也称为 cgroup 子系统）代表一种单一资源：如 CPU 时间或者内存。Linux kernel 提供一系列资源管控器，由 **systemd** 自动挂载。如需参考目前已挂载的资源管控器列表，请参见 **/proc/cgroups**，或使用 **lssubsys** 监控工具。在 Red Hat Enterprise Linux 7 中，**systemd** 默认挂载以下管控器:

Red Hat Enterprise Linux 7 中可用的管控器

- **blkio** —— 对输入／输出访问存取块设备设定权限；
- **cpu** —— 使用 CPU 调度程序让 cgroup 的任务可以存取 CPU。它与 **cpuacct** 管控器一起挂载在同一 mount 上；
- **cpuacct** —— 自动生成 cgroup 中任务占用 CPU 资源的报告。它与 **cpu** 管控器一起挂载在同一 mount 上；
- **cpuset** —— 给 cgroup 中的任务分配独立 CPU（在多芯系统中）和内存节点；
- **devices** —— 允许或禁止 cgroup 中的任务存取设备；
- **freezer** —— 暂停或恢复 cgroup 中的任务；
- **memory** —— 对 cgroup 中的任务可用内存做出限制，并且自动生成任务占用内存资源报告；
- **net_cls** —— 使用等级识别符（classid）标记网络数据包，这让 Linux 流量控制器（**tc** 指令）可以识别来自特定 cgroup 任务的数据包；
- **perf_event** —— 允许使用 **perf** 工具来监控 cgroup；
- **hugetlb** —— 允许使用大篇幅的虚拟内存页，并且给这些内存页强制设定可用资源量。

Linux Kernel 展示了一系列可用 **systemd** 配置的资源管控器可调参数。参数的详细描述请参阅 kernel 文档（kernel 管控器专项介绍 的参考列表）。

1.4. 附加资源

关于单位层级、kernel 资源管控器和 **systemd** 中的资源管控器的更多信息，请参阅以下所列材料：

已安装的文档

与 cgroup 相关的 systemd 文档

以下的 manual page 包含 **systemd** 中统一的 cgroup 层级基本信息：

- **`systemd.resource-control`**(5) —— 描述系统单位共享的资源控制配置选项。
- **`systemd.unit`**(5) —— 描述所有单位配置文件的共同选项。
- **`systemd.slice`**(5) —— 提供 *.slice* 单位的基本信息。
- **`systemd.scope`**(5) —— 提供 *.scope* 单位的基本信息。
- **`systemd.service`**(5) —— 提供 *.service* 单位的基本信息。

kernel 管控器专项介绍

kernel-doc 数据包给所有资源管控器提供了详细文档。此数据包包含在 “可选”（Optional）订阅频道中。订阅 “可选” 频道前，请参阅 Scope of Coverage Details 频道，然后按照 Red Hat 客户门户中《如何使用 Red Hat 订阅管理器（RHSM）获取 “可选” 、 “补充” 频道以及 “ -devel” 数据包？》所述步骤完成。如需从 “可选” 频道中安装 *kernel-doc*，请以 **`root`** 身份输入：

```
yum install kernel-doc
```

安装完成后，下列文件会出现在 **`/usr/share/doc/kernel-doc-<kernel_version>/Documentation/cgroups/`** 目录中：

- **`blkio`** 子系统 —— **`blkio-controller.txt`**
- **`cpuacct`** 子系统 —— **`cpuacct.txt`**
- **`cpuset`** 子系统 ——**`cpusets.txt`**
- **`devices`** 子系统 —— **`devices.txt`**
- **`freezer`** 子系统 —— **`freezer-subsystem.txt`**
- **`memory`** 子系统 —— **`memory.txt`**
- **`net_cls`** 子系统 — **`net_cls.txt`**

另外，关于 **`cpu`** 子系统的更多信息，请参阅下列信息：

- 实时调度 — **`/usr/share/doc/kernel-doc-<kernel_version>/Documentation/scheduler/sched-rt-group.txt`**
- CFS 调度 — **`/usr/share/doc/kernel-doc-<kernel_version>/Documentation/scheduler/sched-bwc.txt`**

在线文档

- 《Red Hat Enterprise Linux 7 系统管理员指南》——《*系统管理员指南*》记录了关于部署、配置和管理 Red Hat Enterprise Linux 7 的相关信息。本指南包含了 systemd 概念的详细释义和使用 systemd 进行服务管理的详尽说明。

- systemd 的 D-Bus API —— D-Bus API 指令的参考资料用于与 **systemd** 互动。

第 2 章 使用控制群组

以下部分将概述与创建、管理控制群组相关的任务。**systemd** 是管理 cgroup 的推荐方式并会在将来版本中被支持，此指南会着重介绍其提供的实用工具。Red Hat Enterprise Linux 的之前版本使用 *libcgroup* 数据包来达到相同目的。此数据包目前仍然可用，以确保后向兼容性（请参阅警告），但 Red Hat Enterprise Linux 之后的版本将不再支持其运行。

2.1. 创建控制群组

从 **systemd** 的角度来看，cgroup 会连接到一个系统单位，此单位可用单位文件进行配置、用 systemd 命令列实用工具进行管理。根据应用的类型，您的资源管理设定可以是 *transient（临时的）* 或者 *persistent（永久的）*。

要为服务创建 **transient cgroup（临时 cgroup）**，请使用 **systemd-run** 指令启动此服务。如此，可以限制此服务在运行时所用资源。对 **systemd** 进行 API 调用，应用程序可以动态创建临时 cgroup。关于如何使用 API，请参阅〈"在线文档"一节〉。服务一旦停止，临时单位就会被自动移除。

要给服务分配 **persistent cgroup（永久 cgroup）**，请对其单位配置文件进行编写。系统重启后，此项配置会被保留，所以它可以用于管理自动启动的服务。请注意，scope 单位不能以此方式创建。

2.1.1. 用 systemd-run 创建临时 cgroup

systemd-run 指令用于创建、启动临时 *service* 或 *scope* 单位，并在此单位中运行自定义指令。在 service 单位中执行的指令在后台非同步启动，它们从 **systemd** 进程中被调用。在 scope 单位中运行的指令直接从 **systemd-run** 进程中启动，因此从调用方继承执行状态。此情况下的执行是同步的。

在一个指定 cgroup 中运行指令，请以 **root** 身份输入：

```
systemd-run --unit=name --scope --slice=slice_name command
```

- *name* 代表您想要此单位被识别的名称。如果 **--unit** 没有被指定，单位名称会自动生成。建议选择一个描述性的名字，因为它将代表 **systemctl** 输出中的单位。在单位运行时期间，此名字需为独一无二的。
- 使用可选的 **--scope** 参数创建临时 *scope* 单位来替代默认创建的 *service* 单位。
- **--slice** 选项，让您新近创建的 *service* 或 *scope* 单位可以成为指定 slice 的一部分。用现存 slice（如 **systemctl -t slice** 输出所示）的名字替代 *slice_name*，或者通过传送一个独有名字来创建新 slice。默认情况下，service 和 scope 做为 **system.slice** 的一部分被创建。
- 用您希望在 service 单位中运行的指令替代 *command*。将此指令放置于 **systemd-run** 句法的最末端。这样，此指令的参数就不会与 **systemd-run** 参数混淆。

除上述选项外，**systemd-run** 也有一些其它可用参数。例如，**--description** 可以创建对单位的描述；service 进程结束后，**--remain-after-exit** 可以收集运行时信息；**--machine** 选项可以在密闭容器中执行指令。更多信息，请参阅 **systemd-run**(1) manual page。

例 2.1. 用 systemd-run 来启动新 service

使用下列指令在名为 **test** 的新 slice 的 service 单位中运行 **top** 实用功能。以 **root** 身份输入：

```
~]# systemd-run --unit=toptest --slice=test top -b
```

如果您正确启动了 service，将显示以下信息：

```
Running as unit toptest.service
```

现在，*toptest.service* 名称可以与 **systemctl** 指令结合，以监控或修改 cgroup。

2.1.2. 创建永久 cgroup

若要在系统启动时，配置一个自动启动的单位，请执行 **systemctl enable** 指令（参见《Red Hat Enterprise Linux 7 系统管理员指南 · *使用 systemd 管理 service* 》）。自动运行此指令会在 **/usr/lib/systemd/system/** 目录中创建单位文件。如要对 cgroup 做出永久改变，请添加或修改其单位文件中的配置参数。更多信息，请参阅〈第 2.3.2 节 "修改单位文件"〉。

2.2. 删除控制群组

临时 cgroup 所包含的进程一旦结束，临时 cgroup 就会被自动释放。通过将 **--remain-after-exit** 选项传递给 **systemd-run**，您可以在其进程结束后，让单位继续运行来收集运行时的信息。如要单位停止运行，请输入：

```
systemctl stop name.service
```

如果您希望一个 service 停止运行，请将 *name* 替换成此 service 的名字。如要终止一个或多个单位中的进程，请以 **root** 身份输入：

```
systemctl kill name.service --kill-who=PID,... --signal=signal
```

用单位名（如 *httpd.service*）替代 *name*。使用 **--kill-who** 从 cgroup 中挑选您希望结束的进程。如要同时终止多项进程，请传送一份 PID 的逗号分隔列表。用您希望发送至指定进程的 POSIX 信号类型替代 *signal*。默认情况下是 *SIGTERM*。更多信息，请参阅 **systemd.kill** manual page。

当单位被禁用并且其配置文件通过运行（下列行）被删除，永久 cgroup 会被释放：

```
systemctl disable name.service
```

此处，*name* 代表您希望禁用的 service 名字。

2.3. 修改 cgroup

所有被 **systemd** 监管的永久单位都在 **/usr/lib/systemd/system/** 目录中有一个单位配置文件。如要修改 service 单位的参数，请修改此配置文件。可以手动完成或者从命令列界面使用 **systemctl set-property** 指令。

2.3.1. 在命令列界面设定参数

systemctl set-property 指令让您可以在应用程序运行时，持续修改资源管控设置。请以 **root** 身份使用下列句法来完成此项操作：

```
systemctl set-property name parameter=value
```

用您希望修改的 systemd 名字来替代 *name*，希望改动的参数名称来替代 *parameter*，希望分配给此参数的新值来替代 *value*。

并非所有单位参数都能在运行时被修改，但是大多数与资源管控相关的参数是可以的。如需要完整列表，请参阅〈第 2.3.2 节 “修改单位文件”〉。提示：**systemctl set-property** 指令让您可以同时修改多项属性，所以相较于单独设定每项属性，推荐您使用此指令。

改动会立即生效并被写入单位文件，并在重启后保留。您可以传递 **--runtime** 选项，让设定变成临时设定。

```
systemctl set-property --runtime name property=value
```

例 2.2. 使用 systemctl set-property

如需使用命令列来限定 *httpd.service* 的 CPU 和内存占用量，请输入：

```
~]# systemctl set-property httpd.service CPUShares=600 MemoryLimit=500M
```

如希望此更改为临时更改，请添加 **--runtime** 选项：

```
~]# systemctl set-property --runtime httpd.service CPUShares=600
MemoryLimit=500M
```

2.3.2. 修改单位文件

systemd service 单位文件提供一系列对资源管理有帮助的高级配置参数。这些参数与必须在 kernel 中启用的 Linux cgroup 管控器通讯。您可以使用这些参数管理 CPU、内存使用量、block IO和更多精细单位的属性。

管理 CPU

cpu 管控器在 kernel 中被默认启动，这可使所有系统 service 的可用 CPU 量相同，而与其所包含进程数量无关。此项默认设定可以使用 **/etc/systemd/system.conf** 配置文件中的 **DefaultControllers** 参数来修改。如需管理 CPU 的分配，请使用单位配置文件 **[Service]** 部分中的下列指令：

CPUShares=*value*

请用 CPU share 的数量代替 *value*。默认值为 1024，您可以增加此数值来给单位分配更多 CPU。此参数默认：**CPUAccounting** 已在单位文件中启用。

CPUShares 参数可以控制 *cpu.shares* 控制群组参数。 请参阅〈Kernel 管控器专项介绍〉对 **cpu** 管控器的描述来查阅其它与 CPU 相关的控制参数。

例 2.3. 限定一个单位的 CPU 可用量

若要为 Apache service 分配 1500 个 CPU share 而不是 1024 个，请修改 **/usr/lib/systemd/system/httpd.service** 单位文件中的 **CPUShares** 设置：

```
[Service]
CPUShares=1500
```

要应用此项修改，请重新载入 systemd 的配置并重启 Apache 来让修改过的 service 文件生效：

```
~]# systemctl daemon-reload
~]# systemctl restart httpd.service
```

内存管理

为限定单位可用内存大小，请使用单位配置文件 **[Service]** 部分中的下列指令：

MemoryLimit=*value*

对 cgroup 中执行的进程设定其可用内存的最大值，并用此值替代 *value*。请以千字节（Kilobyte）、兆字节（Megabyte）、千兆字节（Gigabyte）、太字节（Terabyte）为计量单位并使用 *K*、*M*、*G*、*T* 后缀来表示。同样，**MemoryAccounting** 参数必须在同一单元中启用。

MemoryLimit 参数可以控制 *memory.limit_in_bytes* 控制群组参数。更多信息，请参阅〈Kernel 管控器专项介绍〉中对 **memory** 管控器的描述。

例 2.4. 限制一个单位的可用内存量

若要限定 Apache service 的最大可用内存为 1GB，请修改 **/usr/lib/systemd/system/httpd.service** 单位文件中的 **MemoryLimit** 设定：

```
[Service]
MemoryLimit=1G
```

要应用此项修改，请重新载入 systemd 的配置并重启 Apache 来让修改过的 service 文件生效：

```
~]# systemctl daemon-reload
~]# systemctl restart httpd.service
```

管理 Block IO

如要管理 Block IO，请使用单位配置文件 **[Service]** 部分中的下列指令。下列指令假设 **BlockIOAccounting** 参数已启动：

BlockIOWeight=*value*

为已执行进程选取一个新的整体 block IO 权重，并以此替代 *value*。权重需在 10 到 1000 之间选择，默认值是 1000。

BlockIODeviceWeight=*device_name value*

请为 *device_name* 所指的设备选取 block IO 权重，并以此代替 *value*。用名称或者通向此设备的路径来代替 *device_name*。因为有 **BlockIOWeight**，您可以在 10 到 1000 之间选取权重值。

BlockIOReadBandwidth=*device_name value*

此指令可以为一个单位限定具体带宽。用设备名称或通向块设备节点的路径替换 *device_name*，*value* 代表带宽率。使用 *K*、*M*、*G*、*T* 后缀作为计量单位。没有后缀的值默认单位为 “字节/秒”。

BlockIOWriteBandwidth=*device_name value*

此指令可以给指定设备限定可写带宽。参数与可与 **BlockIOReadBandwidth** 一致。

上述的每一个指令控制一个与之相符的 cgroup 参数。请参阅〈Kernel 管控器专项介绍〉中对 **blkio** 管控器的介绍。

注意

目前，**blkio** 资源管控器暂不支持已缓冲的编写操作。它主要针对直接 I/O，所以已缓冲编写的 service 将忽略 **BlockIOWriteBandwidth** 的限制。另一方面，已缓冲的读取操作是受到支持的，**BlockIOReadBandwidth** 限制对直接读取和已缓冲读取操作均起作用。

例 2.5. 限定一个单位 Block IO 的可用量

如要降低 Apache service 存取 **/home/jdoe/** 目录 block IO 的权重，请将下列字符添加至 **/usr/lib/systemd/system/httpd.service** 单位文件：

```
[Service]
BlockIODeviceWeight=/home/jdoe 750
```

如要设定 Apache 从 **/var/log/** 目录读取的最大带宽为 5MB/秒，请使用下列句法：

```
[Service]
BlockIOReadBandwith=/var/log 5M
```

如要应用此项修改，请重新载入 systemd 的配置并重启 Apache，这样所修改的 service 文件会生效：

```
~]# systemctl daemon-reload
~]# systemctl restart httpd.service
```

管理其它系统资源

另有几种指令，可在单位文件中使用以协助管理资源。

DeviceAllow=*device_name options*

此选项可以控制存取指定设备节点的次数。此处，*device_name* 代表通向设备节点的路径，或者是 **/proc/devices** 中特定的设备组名称。用 **r**、**w** 和 **m** 的组合来替换 **options**，以便单位读取、写入或者创建设备节点。

DevicePolicy=*value*

此处，*value* 是以下三种之一。*strict* ：仅允许 **DeviceAllow** 指定的存取类型；*closed*：允许对标准伪设备的存取，如：/dev/null、/dev/zero、/dev/full、/dev/random 和 /dev/urandom； *auto*：如果不显示 **DeviceAllow**，则允许对所有设备进行存取，此设定为默认设置。

Slice=*slice_name*

请用存放单位的 slice 名称替换 *slice_name*。默认名称是 *system.slice*。scope 单位不能以此方式排列，因为它们已与其父 slice 绑定。

ControlGroupAttribute=*attribute value*

此选项可以设定 Linux cgroup 管控器公开的多项控制群组参数。用您希望修改的低级别 cgroup 参数来替换 *attribute*，用此参数的新值来替换 *value*。更多关于 cgroup 管控器的信息，请参阅〈Kernel 管控器专项介绍〉。

例 2.6. 更改低级别 cgroup 的属性

如果您希望更改 *memory.swappiness* 设置来重新设定 kernel 替换 cgroup 任务所用进程内存的趋势，请参阅〈Kernel 管控器专项介绍〉对内存管控器的介绍。如要将 Apache service 的 *memory.swappiness* 设为 70，请添加下列信息至 **/usr/lib/systemd/system/httpd.service**：

```
[Service]
ControlGroupAttribute=memory.swappiness 70
```

要应用此项修改，请重新载入 systemd 的配置并重启 Apache 来让修改过的 service 文件生效：

```
~]# systemctl daemon-reload
~]# systemctl restart httpd.service
```

2.4. 获得关于控制群组的信息

使用 **systemctl** 指令将系统单位列表并检查它们的状态。**systemd-cgls** 指令可以检查控制群组的层级，**systemd-cgtop** 可以监控控制群组的实时资源消耗。

2.4.1. 将单位列表

使用下列指令将系统中所有被激活单位列表：

```
systemctl list-units
```

list-units 是默认执行选项，也就是说，即便您遗漏了此选项，也不会影响结果：

```
systemctl
UNIT                     LOAD   ACTIVE SUB     DESCRIPTION
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
...
```

以上所列结果包含四项：

- *UNIT* —— 单位名称，也反映单位在 cgroup 树中的位置。 如〈“systemd 的单位类型”一节〉所述，有三种单位类型与资源控制相关：*slice*、*scope* 和 *service*。关于 **systemd** 的单位类型介绍，请参阅《Red Hat Enterprise Linux 7 系统管理员指南 · *使用 systemd 管理 service* 》。
- *LOAD* —— 显示单位配置文件是否被正确装载。如果装载失败，文件会包含 *error* 而不是 *loaded*。其它单位装载状态有：*stub*、*merged* 和 *masked*。
- *ACTIVE* —— 高级单位的激活状态，是 SUB 的一般化。
- *SUB* —— 低级单位的激活状态。可能值的范围取决于单位类型。
- *DESCRIPTION* —— 描述单位内容和性能。

默认情况下，**systemctl** 只会列出被激活的单位（依据 ACTIVE 域中的高级激活状态）。使用 **--all** 选项可以查看未被激活的单位。如要限制结果列表中的信息量，请使用 **--type** (**-t**) 参数，此参数需要单位类型的逗号分隔列表，如：*service* 和 *slice* 或者单位装载状态，如：*loaded* 和 *masked*。

例 2.7. 使用 systemctl list-units

如要查看系统使用的全部 slice 列表，请输入：

```
~]$ systemctl -t slice
```

如要将全部被激活的已屏蔽 service 列表，请输入：

```
~]$ systemctl -t service,masked
```

对您系统中安装的单位文件及其状态列表，请输入：

```
systemctl list-unit-files
```

2.4.2. 查看控制群组的层级

上述指令不会超越单位水平来显示 cgroup 中运行的真正进程。**systemctl** 结果也不会显示单位的层级。您可以使用 **systemd-cgls** 指令，根据 cgroup 将运行的进程分组来同时实现两者。 要显示您系统中的全部 cgroup 层级，请输入：

```
systemd-cgls
```

当 **systemd-cgls** 不带参数发布，它会返回整体 cgroup 层级。cgroup 树的最高层由 slice 构成，外观如下：

```
├─system
│ ├─1 /usr/lib/systemd/systemd --switched-root --system --deserialize 20
│ ...
│
├─user
│ ├─user-1000
│ │ └─ ...
│ ├─user-2000
│ │ └─ ...
│ ...
│
└─machine
  ├─machine-1000
  │ └─ ...
  ...
```

请注意，只有当您运行了虚拟机或者容器的时候，机器 slice 才会出现。更多关于 cgroup 树的信息，请参阅〈“systemd 的单位类型”一节〉。

如要减少 **systemd-cgls** 的输出并查看层级的特定一部分，请执行：

```
systemd-cgls name
```

请用您希望检查的资源管控器的名字替换 *name*。

或者，使用 **systemctl status** 来显示系统单位的详细信息。cgroup 的子树是此指令结果的一部分。

```
systemctl status name
```

更多关于 **systemctl status** 的信息，请参阅《Red Hat Enterprise Linux 7 系统管理员指南 · *使用 systemd 管理 service*》。

例 2.8. 查看控制群组的层级

如要查看 **memory** 资源管控器的 cgroup 树，请执行：

```
~]$ systemd-cgls memory
memory:
├    1 /usr/lib/systemd/systemd --switched-root --system --deserialize 23
├  475 /usr/lib/systemd/systemd-journald
...
```

以上指令的结果将列出与所选管控器互动的 service。另一种方法是查看 cgroup 树的一部分来查阅 service、slice 或者 scope 单位：

```
~]# systemctl status httpd.service
httpd.service - The Apache HTTP Server
   Loaded: loaded (/usr/lib/systemd/system/httpd.service; enabled)
   Active: active (running) since Sun 2014-03-23 08:01:14 MDT; 33min ago
  Process: 3385 ExecReload=/usr/sbin/httpd $OPTIONS -k graceful
(code=exited, status=0/SUCCESS)
 Main PID: 1205 (httpd)
   Status: "Total requests: 0; Current requests/sec: 0; Current traffic:
0 B/sec"
   CGroup: /system.slice/httpd.service
           ├─1205 /usr/sbin/httpd -DFOREGROUND
           ├─3387 /usr/sbin/httpd -DFOREGROUND
           ├─3388 /usr/sbin/httpd -DFOREGROUND
           ├─3389 /usr/sbin/httpd -DFOREGROUND
           ├─3390 /usr/sbin/httpd -DFOREGROUND
           └─3391 /usr/sbin/httpd -DFOREGROUND

...
```

除了上述工具，**systemd** 也提供了专门监控 Linux 的 **machinectl** 指令。

2.4.3. 查看资源管控器

上述的 **systemctl** 指令可以监控高级单位层级，但是不能显示 Linux kernel 的资源管控器被哪项进程使用。这些信息存储在专门的进程文件中，如要查阅这些文件，请以 **root** 身份输入：

```
cat proc/PID/cgroup
```

PID 代表您希望查看的进程 ID。默认情况下，此列表对所有 **systemd** 启动的单位一致，因为它自动挂载所有默认管控器。请参考下列示例：

```
~]# cat proc/27/cgroup
10:hugetlb:/
9:perf_event:/
```

```
8:blkio:/
7:net_cls:/
6:freezer:/
5:devices:/
4:memory:/
3:cpuacct,cpu:/
2:cpuset:/
1:name=systemd:/
```

通过检查此文件，您可以确定进程是否被放置在 systemd 单位文件规范所定义的所需 cgroup 中。

2.4.4. 监控资源消耗量

systemd-cgls 指令给 cgroup 层级提供了静态数据快照。要查看按资源使用量（CPU、内存和 IO）排序的、正在运行的 cgroup 动态描述请使用：

```
systemd-cgtop
```

systemd-cgtop 提供的统计数据和控制选项与 **top** 实用工具所提供的相近。更多信息，请参阅 **systemd-cgtop**(1) manual page。

2.5. 附加资源

关于如何使用 **systemd** 及其相关工具管理 Red Hat Enterprise Linux 系统资源，请参阅以下资料：

已安装文件

与 cgroup 相关的 Systemd 工具帮助页面

- **systemd-run**(1) —— 此 manual page 列出了 **systemd-run** 实用工具的全部命令列选项。
- **systemctl**(1) —— **systemctl** 实用工具的 manual page 列出了可用选项及指令。
- **systemd-cgls**(1) —— 此 manual page 列出了 **systemd-cgls** 实用工具的全部命令列选项。
- **systemd-cgtop**(1) —— 此 manual page 包含了 **systemd-cgtop** 实用工具的全部命令列选项。
- **machinectl**(1) —— 此 manual page 列出了 **machinectl** 实用工具的全部命令列选项。
- **systemd.kill**(5) —— 此 manual page 为系统单位提供了终止配置选项的概述。

Kernel 管控器专项介绍

kernel-doc 数据包给所有资源管控器提供了详细文档。此数据包包含在 “可选”（Optional）订阅频道中。订阅 “可选” 频道前，请参阅 Scope of Coverage Details 频道，然后按照 Red Hat 客户门户中《如何使用 Red Hat 订阅管理器（RHSM）获取 “可选”、“补充” 频道以及 “ -devel” 数据包？》所述步骤完成。如需从 “可选” 频道中安装 *kernel-doc*，请以 **root** 身份输入：

```
yum install kernel-doc
```

安装完成后，下列文件将出现在 **/usr/share/doc/kernel-doc-<*kernel_version*>/Documentation/cgroups/** 目录下：

- **blkio** 子系统 —— **blkio-controller.txt**

- **cpuacct** 子系统 —— **cpuacct.txt**
- **cpuset** 子系统 —— **cpusets.txt**
- **devices** 子系统 —— **devices.txt**
- **freezer** 子系统 —— **freezer-subsystem.txt**
- **memory** 子系统 —— **memory.txt**
- **net_cls** 子系统 —— **net_cls.txt**

关于 **cpu** 子系统的更多信息，请参阅下列文件：

- 实时调度 — **/usr/share/doc/kernel-doc-<*kernel_version*>/Documentation/scheduler/sched-rt-group.txt**
- CFS 调度 — **/usr/share/doc/kernel-doc-<*kernel_version*>/Documentation/scheduler/sched-bwc.txt**

在线文档

- Red Hat Enterprise Linux 7 系统管理员指南 ——《*系统管理员指南*》记录了关于部署、配置和管理 Red Hat Enterprise Linux 7 的相关信息。 这要求系统管理员对系统有基本了解。
- systemd 的 D-Bus API —— D-Bus API 指令存取 systemd 的参考。

第 3 章 使用 libcgroup 工具

libcgroup 数据包, 在 Red Hat Enterprise Linux 之前版本中是管理 cgroup 的主要工具，但现已过时。为避免冲突，请不要将 *libcgroup* 工具应用于默认资源管控器（列于 Red Hat Enterprise Linux 7 中可用的管控器），它们现在是 **systemd** 的专属域。这就让 *libcgroup* 工具的应用空间变得有限，只有当您需要管理目前不受 **systemd** 支持的控制器时，请再使用它，例如 *net_prio*。

以下章节将描述：与层级的默认系统没有冲突时，如何在相关情景中使用 *libcgroup* 工具。

注意

使用 *libcgroup* 工具前，请先确保 *libcgroup* 和 *libcgroup-tools* 数据包已安装在系统上。安装方法：请以 **root** 身份运行：

```
~]# yum install libcgroup
~]# yum install libcgroup-tools
```

注意

net_prio 管控器没有像其它管控器一样被编译在 kernel 中，它是一个在挂载前必须被装载的模块。如要装载这一模块，请以 **root** 身份输入：

```
modprobe netprio_cgroup
```

3.1. 挂载层级

如需使用非自动挂载的 kernel 资源管控器，您需要创建一个包含这一管控器的层级。通过编写 **/etc/cgconfig.conf** 的 **mount** 部分来添加或者分离层级。这一方法可让管控器附载持续，也就是说系统重启后，您的设定也将被保留并运作。或者，您可以使用 **mount** 指令为当前会话创建临时挂载。

使用 cgconfig 服务

通过 *libcgroup-tools* 数据包安装的 **cgconfig** 服务可以让附加资源的管控器挂载层级。默认情况下，这一服务不会自动启动。当您启动 **cgconfig** 时，它会应用 **/etc/cgconfig.conf** 配置文件中的设置。因此，从会话到会话，配置会被重建，并且持续。请注意，如果您终止 **cgconfig**，那它之前挂载的层级都将被卸载。

通过 *libcgroup* 数据包安装的默认 **/etc/cgconfig.conf** 文件不包含任何配置设置，仅会有 **systemd** 自动挂载主要资源管控器的信息。

/etc/cgconfig.conf 可创建三种条目 —— *mount*、*group* 和 *template*。mount 条目用于创建层级并将层级挂载为虚拟文件系统，并且将管控器附加到这些层级中。Red Hat Enterprise Linux 7 中， 默认层级会被自动挂载到 **/sys/fs/cgroup/** 目录，因此，**cgconfig** 被用来单独附载非默认的管控器。mount 条目通过以下句法定义：

```
mount {
    controller_name = /sys/fs/cgroup/controller_name;
    …
}
```

用您希望挂载到层级的 kernel 资源管控器名称来替换*controller_name*。示例请参见 例 3.1 “创建挂载条目”。

例 3.1. 创建挂载条目

如要将 **net_prio** 管控器附加到默认 cgroup 树中，请将下列字符添加到 **/etc/cgconfig.conf** 配置文件：

```
mount {
    net_prio = /sys/fs/cgroup/net_prio;
}
```

然后重启 **cgconfig** 服务来应用这些设置：

```
systemctl restart cgconfig.service
```

/etc/cgconfig.conf 中的组条目可以用来设定资源管控器的参数。更多组条目的信息，请参阅〈第 3.5 节 “设定 cgroup 参数”〉。

/etc/cgconfig.conf 中的样板条目可以用来创建组定义，这个组定义将应用于所有进程。

使用挂载指令

mount 指令可以用于临时挂载层级。如需这样做，请先在 **/sys/fs/cgroup/** 目录中创建一个挂载点，**systemd** 将在此目录中挂载主要资源管控器。请以 **root** 身份输入:

```
mkdir /sys/fs/cgroup/name
```

请用新挂载目标的名称替代 *name*，通常是用管控器的名称。接下来，执行 **mount** 指令来挂载层级，同时附载一个或更多子系统。以 **root** 身份来输入:

```
mount -t cgroup -o controller_name none /sys/fs/cgroup/controller_name
```

用管控器的名称取代 *controller_name* 来具体说明被挂载的设备和目标文件夹。**-t cgroup** 参数可以明确挂载的类型。

例 3.2. 使用挂载指令来附加控制器

如要使用 **mount** 指令给 **net_prio** 管控器挂载层级，请先创建挂载节点：

```
~]# mkdir /sys/fs/cgroup/net_prio
```

然后将 **net_prio** 挂载到您之前创建的目标中。

```
~]# mount -t cgroup -o net_prio none /sys/fs/cgroup/net_prio
```

通过 **lssubsys** 指令（参见〈"将管控器列表"一节〉），将所有可用层级沿当前挂载节点列表，您就可以验证是否正确附加了层级：

```
~]# lssubsys -am
cpuset /sys/fs/cgroup/cpuset
cpu,cpuacct /sys/fs/cgroup/cpu,cpuacct
memory /sys/fs/cgroup/memory
devices /sys/fs/cgroup/devices
freezer /sys/fs/cgroup/freezer
net_cls /sys/fs/cgroup/net_cls
blkio /sys/fs/cgroup/blkio
perf_event /sys/fs/cgroup/perf_event
hugetlb /sys/fs/cgroup/hugetlb
net_prio /sys/fs/cgroup/net_prio
```

3.2. 卸载层级

如果您通过编写 **/etc/cgconfig.conf** 配置文件挂载了一个层级，那您可以很简单地，从此配置文件的 *mount* 部分移除配置指令来卸载它。然后重启服务以应用新的配置。

同样，您可以以 **root** 身份执行以下指令来卸载层级：

```
~]# umount /sys/fs/cgroup/controller_name
```

用包含您希望分离的资源管控器的层级名称来替换 *controller_name*。

警告

请确保只使用 **umount** 来移除您自己手动挂载的层级。分离包含默认管控器（参见 Red Hat Enterprise Linux 7 中可用的管控器）的层级很可能会导致异常，并需要重启。

3.3. 创建控制群组

在您自己创建的层级中，您可以使用 **cgcreate** 指令来创建临时 cgroup。**cgcreate** 的句法是:

```
cgcreate -t uid:gid -a uid:gid -g controllers:path
```

其中：

- **-t**（可选）—— 指定一个用户（通过用户 ID：uid）和群组（通过群组 ID：gid）来拥有此 cgroup 的 **tasks** 伪文件。此用户可在该 cgroup 中添加任务。

注意

请注意，从 cgroup 中移除进程的唯一方法是将进程移至另一个 cgroup。如想要移除进程，用户必须拥有 "*目标*" cgroup 的写入权限；但是源 cgroup 的写入权限并不重要。

- **-a**（可选）—— 指定一个用户（通过用户 ID：uid）和群组（通过群组 ID：gid）来拥有此 cgroup 的全部

伪文件而不是 **tasks** 。此用户可以修改 cgroup 中任务存取系统资源的权限。

- **-g** —— 指定 cgroup 应该被建于其中的层级，类似于"*管控器*"和这些层级的列表（以逗号分隔）。管控器的此项列表之后是一个冒号以及相对层级的子群组"*路径*"。请不要将层级挂载点包含于路径中。

同一层级中的所有 cgroup 有同一个管控器，所以子群组与其父群组的管控器也相同。

或者，您可以使用 **mkdir** 指令直接给此 cgroup 创建一个子 cgroup：

```
~]# mkdir /sys/fs/cgroup/controller/name/child_name
```

例如：

```
~]# mkdir /sys/fs/cgroup/net_prio/lab1/group1
```

3.4. 删除控制群组

可以用与 **cgcreate** 句法相似的 **cgdelete** 指令来移除 cgroup。请以 **root** 身份运行以下指令:

```
cgdelete controllers:path
```

其中：

- *controller* 是管控器的逗号分隔清单。
- *path* 是与该层级的根相对的 cgroup 路径。

例如：

```
~]# cgdelete net_prio:/test-subgroup
```

指定 **-r** 选项时，**cgdelete** 也可以递归式移除所有子群组。

请注意，当您删除一个 cgroup，它的全部进程会移动到其父群组。

3.5. 设定 cgroup 参数

如要修改控制群组参数，您可以编写 **/etc/cgconfig.conf** 或者使用 **cgset** 指令。 重启后，对 **/etc/cgconfig.conf** 的更改也将保留并运行，但是**cgset** 仅能对当前会话的 cgroup 参数做出修改。

修改 /etc/cgconfig.conf

您可以在 **/etc/cgconfig.conf** 的 *Groups* 部分设定管控器参数。 组条目使用以下句法定义：

```
group name {
[permissions]
    controller {
        param_name = param_value;
        …
    }
    …
}
```

请用您 cgroup 的名字替代 *name*， *controller* 代表您希望修改的管控器名称。这应当是一个您自己挂载的管控器，而不是 **systemd** 自动挂载的默认管控器。用您希望更改的管控器参数和其新数值替代 *param_name* 和 *param_value*。请注意，**permissions** 部分是可选择项。如要给组条目定义权限，请使用以下句法：

```
perm {
    task {
        uid = task_user;
        gid = task_group;
    }
    admin {
       uid = admin_name;
       gid = admin_group;
    }
}
```

注意

请重启 **cgconfig** 服务，让 **/etc/cgconfig.conf** 的更改生效。重启此服务会重建配置文件中指定的层级，但并不会影响所有挂载层级。您可以通过执行 **systemctl restart** 指令来重启服务，但是，建议您先停止 **cgconfig**：

```
~]# systemctl stop cgconfig
```

然后打开并编写配置文件。保存更改后，您可以用以下指令再次启动 **cgconfig**：

```
~]# systemctl start cgconfig
```

使用 cgset 指令

获得修改相关 cgroup 的权限后，请运行用户账户中的 **cgset** 指令来设定管控器参数。请仅对手动挂载的管控器使用此指令。

cgset 的句法为：

```
cgset -r parameter=value path_to_cgroup
```

其中：

- *parameter* 是要设定的参数，它与给定 cgroup 目录中的文件对应；
- *value* 是参数值；
- *path_to_cgroup* 是“*与层级的根相对*”的 cgroup 路径。

cgset 设定的值可能会受限于一个特定层级所设定的更高值。例如，在一个系统中，如果 **group1** 被限定仅可使用 CPU 0，那您就不能设定 **group1/subgroup1** 使用 CPU 0 和 1，或者仅使用 CPU 1。

您也可以使用 **cgset** 将一个 cgroup 的参数复制到另一个已有 cgroup 中。使用 **cgset** 复制参数的句法是:

```
cgset --copy-from path_to_source_cgroup path_to_target_cgroup
```

其中：

- *path_to_source_cgroup* 是要复制其参数的 cgroup 路径，相对层级的根群组；
- *path_to_target_cgroup* 是目标 cgroup 的路径，相对层级的根群组。

3.6. 将进程移至控制群组

您可以运行 **cgclassify** 指令将进程移动到 cgroup 中：

```
cgclassify -g controllers:path_to_cgroup pidlist
```

其中：

- *controllers* 是资源控制器列表，以逗号分隔；或者使用 ***** 来启动与所用可用子系统相关的层级中的进程。请注意，如果几个 cgroup 的名称相同，**-g** 选项会将进程移至这些群组的每一个。
- *path_to_cgroup* 是层级中 cgroup 的路径；
- *pidlist* 是 *process identifier*（PID）的列表，以空格隔开。

如果没有明确 **-g** 选项，**cgclassify** 会自动搜索 **/etc/cgrules.conf**，并且使用第一个适用配置行。根据此配置行，**cgclassify** 会确定进程将移至的层级和 cgroup。请注意，为确保移动成功，目标层级必须存在。**/etc/cgrules.conf** 指定的子系统也必须被合理配置来回应 **/etc/cgconfig.conf** 中的层级。

您还可以在 *pid* 前面添加 **--sticky** 选项以保证所有子进程位于同一 cgroup 中。如果您没有设定这个选项且 **cgred** 服务正在运行，则子进程会根据 **/etc/cgrules.conf** 中的设置被分配到 cgroup 中。而该进程本身仍保留在启动它的 cgroup 中。

您还可以使用 **cgred** 服务（**cgrulesengd** 后台驻留程序启动服务），此服务会根据 **/etc/cgrules.conf** 文件中设定的参数将任务移至 cgroup。请仅将 **cgred** 用于管理手动附加的管控器。**/etc/cgrules.conf** 文件条目的形式可以是以下两种之一：

- *user subsystems control_group*；
- *user*:*command subsystems control_group*。

例如：

```
maria   net_prio  /usergroup/staff
```

此条目指定任何属于名为 **maria** 用户的进程，都可以根据 **/usergroup/staff** cgroup 指定的参数存取 **devices** 子系统。要将特定指令与具体 cgroup 关联，请添加 *command* 参数，如下：

```
maria:ftp  devices  /usergroup/staff/ftp
```

此条目现指定，当名为 **maria** 的用户使用 **ftp** 指令时，该进程将被自动移动到包含 **devices** 子系统的层级的 **/usergroup/staff/ftp** cgroup 中。然而，请注意：只有在符合适当条件后，后台驻留程序才可将该进程移动到 cgroup 中。因此，**ftp** 进程可能会在错误的群组中短暂运行。另外，如果该进程在错误群组中迅速生成子进程，这些子进程可能不会被移动。

/etc/cgrules.conf 文件中的条目可包括以下额外符号：

- **@** —— 被用作 *user* 前缀时，它代表一个群组而不是单独用户。例如：**@admins** 表示 **admins** 组群中的所有用户。

- ***** —— 代表"所有"。例如：**subsystem** 字段中的 ***** 代表所有子系统。
- **%** —— 代表与上一行相同的项目。例如：

```
@adminstaff net_prio   /admingroup
@labstaff %   %
```

3.7. 启动控制群组的进程

您可以通过运行 **cgexec** 指令在手动创建的 cgroup 中启动进程。**cgexec** 的语法为：

```
cgexec -g controllers:path_to_cgroup command arguments
```

其中：

- *controllers* 是管控器列表，以逗号分隔；或者使用 ***** 来启动与所用可用子系统相关的层级中的进程。请注意，与〈第 3.5 节 "设定 cgroup 参数"〉所描述的 **cgset** 指令一样，如果几个 cgroup 的名称相同，**-g** 选项会在每一个 cgroup 中都创建进程；
- *path_to_cgroup* 是与层级相对的 cgroup 路径；
- *command* 是要在该 cgroup 中执行的指令；
- *arguments* 是该指令的所有参数。

您也可以在 *command* 前添加 **--sticky** 选项，以便让所有子进程位于同一个 cgroup 中。如果您没有设定此选项且 **cgred** 后台驻留程序正在运行，子进程将会根据 **/etc/cgrules.conf** 中的设定被分配到 cgroup 中。 而进程本身会保留在启动它的 cgroup 中。

3.8. 获区关于控制群组的信息

libcgroup-tools 数据包包含一些使用工具，这些工具用于获取与管控器、控制群组以及控制群组参数有关的信息。

将管控器列表

如要查找 kernel 中可用的管控器，并了解这些管控器如何被一起挂载至层级，请执行：

```
cat /proc/cgroups
```

如要查找特定子系统的挂载点，请执行下列指令：

```
lssubsys -m controllers
```

此处，*controllers* 代表您感兴趣的子系统列表。请注意，**lssubsys -m** 指令仅返回每一个层级的顶级挂载点。

查找控制群组

如要将系统中的 cgroup 列表，请以 **root** 身份执行：

```
lscgroup
```

如要限定一个层级的输出，请以 ***controller*:*path*** 的格式指定管控器和路径。例如：

```
~]$ lscgroup cpuset:adminusers
```

以上指令仅列出了层级（附加了**cpuset** 控制器的层级）中 **adminusers** cgroup 的子群组。

显示控制群组的参数

如要显示一个 cgroup 的参数，请运行：

```
~]$ cgget -r parameter list_of_cgroups
```

此处，*parameter* 是一个包含管控器值的伪文件，*list_of_cgroups* 是管控器列表（以逗号分隔）。

如果您不知道实际参数的名称，请使用与以下相似的指令：

```
~]$ cgget -g cpuset /
```

3.9. 附加资源

如需要 cgroup 指令的最终文档，请在 *libcgroup* 数据包提供的 manual page 中查找。

已安装的文档

与 libcgroup 相关的 manual page

- **cgclassify**(1) —— **cgclassify** 指令用于将正在运行的任务移至一个或多个 cgroup 中。
- **cgclear**(1) —— **cgclear** 指令用于删除层级中的全部 cgroup。

  cgconfig.conf(5) —— cgroup 在 **cgconfig.conf** 文件中被定义。
- **cgconfigparser**(8) —— **cgconfigparser** 指令用于解析 **cgconfig.conf** 文件并且挂载层级。
- **cgcreate**(1) —— **cgcreate** 指令用于在层级中创建新的 cgroup。
- **cgdelete**(1) —— **cgdelete** 指令用于移除指定的 cgroup。
- **cgexec**(1) —— **cgexec** 指令用于在指定的 cgroup 中运行任务。
- **cgget**(1) —— **cgget** 指令用于显示 cgroup 参数。
- **cgsnapshot**(1) —— **cgsnapshot** 指令用于从现存的子系统中生成配置文件。
- **cgred.conf**(5) —— **cgred.conf** 是 **cgred** 服务的配置文件。
- **cgrules.conf**(5) —— **cgrules.conf** 包含可以确定任务何时归属于某一 cgroup 的规则。
- **cgrulesengd**(8) —— **cgrulesengd** 服务用于将任务分配到 cgroup。
- **cgset**(1) —— **cgset** 指令用于为 cgroup 设定参数。
- **lscgroup**(1) —— **lscgroup** 指令用于将层级中的 cgroups 列表。

- **`lssubsys`**(1) — **`lssubsys`** 指令将包含特定子系统的层级列表。

第 4 章 控制群组应用示例

本章将针对如何使用 cgroup 给出应用示例。

4.1. 定义数据库 I/O 的优先级

在数据库服务器专用的虚拟机内部运行数据库服务器实例，让您可以根据数据库的优先级来为其分配资源。请参考下列示例：系统在两个 KVM 虚拟机内部运行两个数据库服务器。一个数据库的优先级较高，另一个较低。当两个数据库服务器同时运行，I/O 吞吐量会降低来均等地容纳两个数据库的请求；如图 4.1 “不根据优先级分配资源时的 I/O 吞吐量” 所示：一旦优先级低的数据库启动（约在时间轴的 45 处），分配给两个服务器的 I/O 吞吐量是相同的。

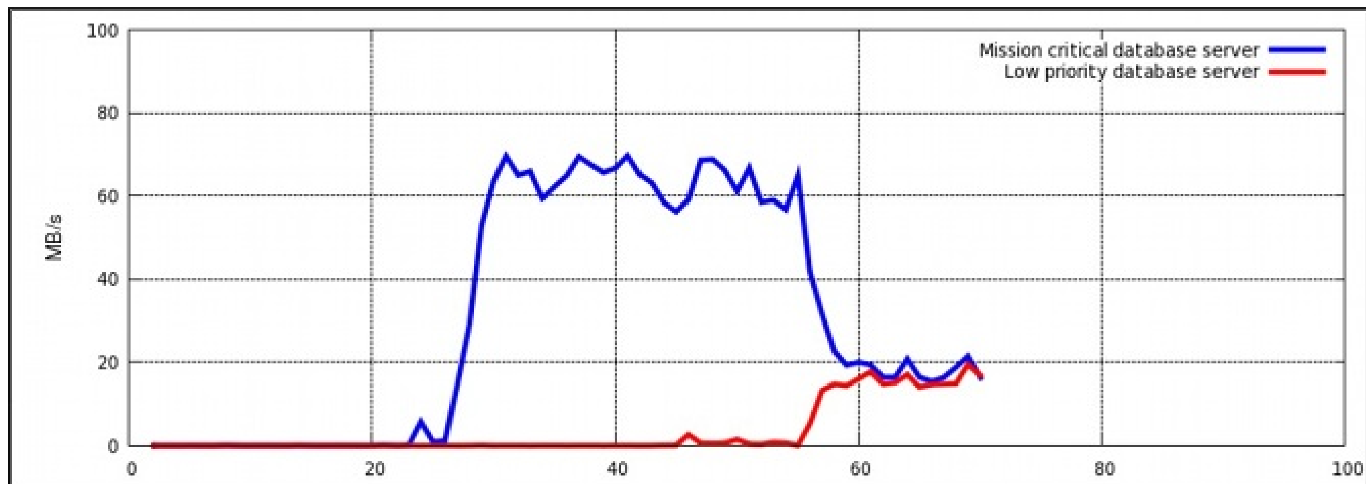

图 4.1. 不根据优先级分配资源时的 I/O 吞吐量

为能优先处理来自优先级高的数据库服务器请求，可将此服务器分配给一个 I/O 操作预留量高的 cgroup，而优先级低的数据库服务器可以分配给一个 I/O 操作预留量少的 cgroup。可按照以下步骤过程 4.1, “I/O 吞吐量优先化”来完成此操作，这些步骤将全部在主机系统上执行。

过程 4.1. I/O 吞吐量优先化

1. 请确保两项服务的所用资源统计功能，处于开启状态：

```
~]# systemctl set-property db1.service BlockIOAccounting=true
~]# systemctl set-property db2.service BlockIOAccounting=true
```

2. 如果将高优先级和低优先级服务的比率设定为 10：1 ，那么在这些服务单位中运行的进程将只能使用可用资源：

```
~]# systemctl set-property db1.service BlockIOWeight=1000
~]# systemctl set-property db2.service BlockIOWeight=100
```

图 4.2 “根据优先级分配资源时的 I/O 吞吐量” 显示了优先处理优先级高的数据库请求，而限制优先级低的数据库的情况。一旦数据库服务器移至恰当的 cgroup（约在时间轴的 75 处），I/O 吞吐量就会在两个服务器间按照 10：1 的比率分配。

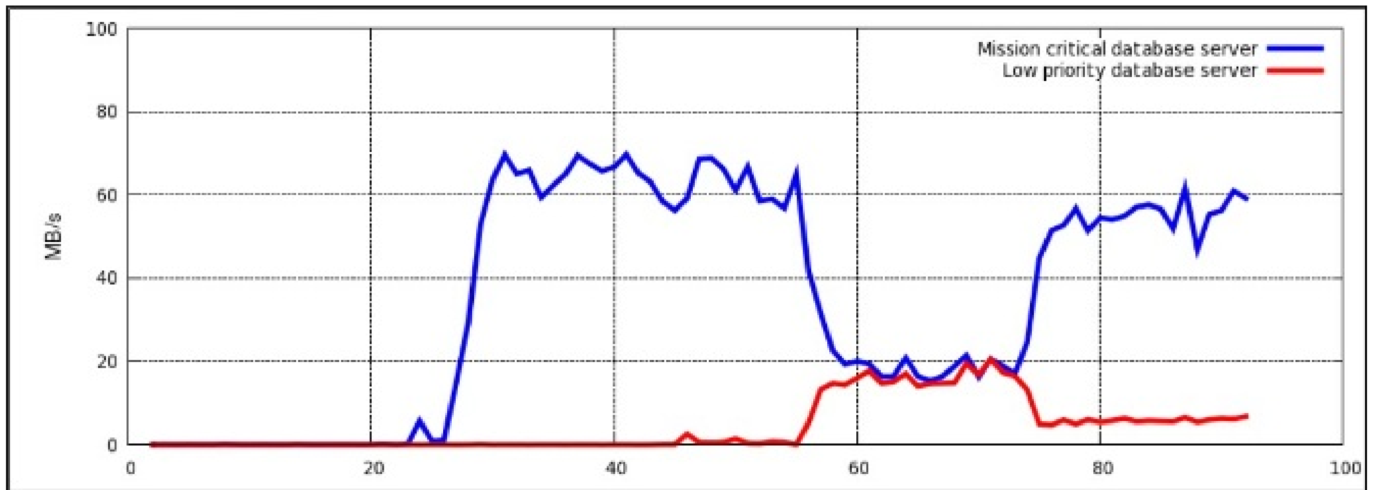

图 4.2. 根据优先级分配资源时的 I/O 吞吐量

或者，块设备 I/O 流量调节功能，可对优先级低的数据库限定其读写操作量。更多信息，请参阅 〈kernel 管控器专项介绍〉对 **blkio** 管控器的介绍。

4.2. 定义网络流量的优先级

在单一服务器系统中运行多项与网络相关的服务时，定义这些服务的网络优先级是很重要的。定义优先级可以保证源自特定服务的数据包比源自其它服务的数据包享有更高优先级。例如，当一台服务器系统同时起到 NFS 服务器和 Samba 服务器的作用时，优先级就显得尤为重要。NFS 必须享有高优先权，因为用户会预期较高吞吐量。Samba 的优先级可以较低，以确保 NFS 服务器有更佳表现。

net_prio 管控器可以用来为 cgroup 中的进程设定网络优先级。之后，优先级会被转译为 Type Of Service（TOS，服务类型）比特，并嵌入每一个数据包中。请参照过程 4.2, “为共享服务的文件设定网络优先级” 中的步骤给两份文件共享的服务（NFS 和 Samba）配置优先级。

过程 4.2. 为共享服务的文件设定网络优先级

1. **net_prio** 管控器并未编译进 kernel，它是一个必须手动装载的模块。如需装载，请输入：

   ```
   ~]# modprobe netprio_cgroup
   ```

2. 请将 **net_prio** 子系统附加到 **/cgroup/net_prio** cgroup 中：

   ```
   ~]# mkdir sys/fs/cgroup/net_prio
   ~]# mount -t cgroup -o net_prio none sys/fs/cgroup/net_prio
   ```

3. 请为各项服务创建其 cgroup：

   ```
   ~]# mkdir sys/fs/cgroup/net_prio/nfs_high
   ~]# mkdir sys/fs/cgroup/net_prio/samba_low
   ```

4. 如希望 **nfs** 服务被自动移至 **nfs_high** cgroup，请将下列行添至 **/etc/sysconfig/nfs** 文件:

   ```
   CGROUP_DAEMON="net_prio:nfs_high"
   ```

   此配置可确保 **nfs** 服务启动或重启时，**nfs** 服务进程已被移至 **nfs_high** cgroup。

5. **smbd** 后台驻留程序在 **/etc/sysconfig** 目录中没有配置文件。为实现将 **smbd** 后台驻留程序自动移至 **samba_low** cgroup，请添加下列行至 **/etc/cgrules.conf** 文件：

```
*:smbd                net_prio                samba_low
```

请注意：此规则会将每一个 **smbd** 后台驻留程序（不仅仅是 **/usr/sbin/smbd**）移至 **samba_low** cgroup 。

您可以用相似的方式为 **nmbd** 和 **winbindd** 后台驻留程序定义规则，将它们移至 **samba_low** cgroup。

6. 请启动 **cgred** 服务，以载入之前步骤的配置：

```
~]# systemctl start cgred
Starting CGroup Rules Engine Daemon:                       [  OK  ]
```

7. 至于此示例的目的，让我们假设两项服务都使用 **eth1** 网络接口；给每一个 cgroup 定义优先级：**1** 表示优先级低，**10** 表示优先级高：

```
~]# echo "eth1 1" >
/sys/fs/cgroup/net_prio/samba_low/net_prio.ifpriomap
~]# echo "eth1 10" >
/sys/fs/cgroup/net_prio/nfs_high/net_prio.ifpriomap
```

8. 请启动 **nfs** 和 **smb** 服务以检查各自的进程是否已被移至正确的 cgroup：

```
~]# systemctl start smb
Starting SMB services:                                     [  OK  ]
~]# cat /sys/fs/cgroup/net_prio/samba_low/tasks
16122
16124
~]# systemctl start nfs
Starting NFS services:                                     [  OK  ]
Starting NFS quotas:                                       [  OK  ]
Starting NFS mountd:                                       [  OK  ]
Stopping RPC idmapd:                                       [  OK  ]
Starting RPC idmapd:                                       [  OK  ]
Starting NFS daemon:                                       [  OK  ]
~]# cat sys/fs/cgroup/net_prio/nfs_high/tasks
16321
16325
16376
```

现在，源自于 NFS 的网络信息传输比源自于 Samba 的信息传输有更高的优先级。

与过程 4.2, “为共享服务的文件设定网络优先级” 相似，**net_prio** 子系统可以用来为客户应用程序（如火狐）设定网络优先级。

附录 A. 子系统和可调参数

“*子系统*” 是识别 cgroup 的 kernel 模块。通常，它们被看做资源管控器，为不同 cgroup 分配不同级别的系统资源。但是当不同的进程组需要被区别对待时，可以改写子系统与 kernel 间的互动。用于开发新子系统的 “*应用程序编程界面*”（API）记载于 kernel 文档的 **`cgroups.txt`** 中，该文件安装在您系统的 **`/usr/share/doc/kernel-doc-`*`kernel-version`*`/Documentation/cgroups/`** 中（由 *kernel-doc* 软件包提供）。cgroup 文档的最新版本可在 http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt 中找到。请注意：最新文档所述功能可能不与您系统中已安装的 kernel 功能匹配。

包含 cgroup 子系统参数的“*状态对象*”在 cgroup 的虚拟文件系统中，被描述为 “*伪文件*”。这些伪文件可由 shell 命令或者与其对等的系统呼叫来操作。例如：**`cpuset.cpus`** 是一份指定 cgroup可以存取哪个 CPU 的伪文件。如果 **`/cgroup/cpuset/webserver`** 是系统中运行的网页服务器的 cgroup，那么以下命令会被执行：

```
~]# echo 0,2 > /cgroup/cpuset/webserver/cpuset.cpus
```

因为 **`cpuset.cpus`** 伪文件中写入了 **`0,2`** 值，所以任何任务的 PID 一旦被列入 **`/cgroup/cpuset/webserver/tasks/`**，那么此任务将仅能在系统中使用 CPU 0 和 CPU 2。

A.1. blkio

块 I/O（**`blkio`**）子系统可以控制并监控 cgroup 中的任务对块设备 I/O 的存取。对一些伪文件写入值可以限制存取次数或带宽，从伪文件中读取值可以获得关于 I/O 操作的信息。

`blkio` 子系统给出两种方式来控制对 I/O 的存取：

- “*权重分配*” — 用于完全公平列队 I/O 调度程序 （Completely Fair Queuing I/O scheduler），用此方法，让您可以给指定的 cgroup 设定权重。这意味着每个 cgroup 都有一个预留的 I/O 操作设定比例（根据 cgroup 的权重）。详情请参阅〈第 A.1.1 节 “权重分配的可调参数”〉。
- “*I/O 节流（上限）*” — 当一个指定设备执行 I/O 操作时，此方法可为其操作次数设定上限。这意味着一个设备的 “*读*” 或者 “*写*” 的操作次数是可以限定的。详情请参阅第 A.1.2 节 “I/O 节流可调参数”。

重要

目前，block I/O 子系统不支持已缓冲的 “*写*” 操作。虽然可以支持已缓冲的 “*读*” 操作，但它主要针对直接 I/O。

A.1.1. 权重分配的可调参数

blkio.weight

此参数用于指定一个 cgroup 在默认情况下可存取块 I/O 的相对比例（*加权*），范围是 **`100`** 到 **`1000`**。该值可被指定设备的 **`blkio.weight_device`** 参数覆盖。例如：如要将 cgroup 存取块设备的默认权重设定为 **`500`**，请运行：

```
echo 500 > blkio.weight
```

blkio.weight_device

此参数用于设定 cgroup 中指定设备 I/O 存取的相对比例（*加权*），范围是 **`100`** 到 **`1000`**。对于指定的设备，此参数值可覆盖 **`blkio.weight`** 参数值。值的格式为 *major:minor weight*。其中 *major* 和

minor 是〈*Linux 分配的设备*〉所指定的设备类型和节点数，我们也称之为〈*Linux 设备列表*〉，可从 http://www.kernel.org/doc/Documentation/devices.txt 中找到。例如：如果设定 cgroup 访问 **/dev/sda** 的权重为 **500**，请运行：

```
echo 8:0 500 > blkio.weight_device
```

在〈*Linux 分配的设备*〉标记法中，**8:0** 代表 **/dev/sda**。

A.1.2. I/O 节流可调参数

blkio.throttle.read_bps_device

此参数用于设定设备执行“*读*”操作字节的上限。“*读*”的操作率以每秒的字节数来限定。条目有三种字段：*major*、*minor* 和 *bytes_per_second*。*major* 和 *minor* 是〈*Linux 分配的设备*〉所指定的设备类型和节点数。*bytes_per_second* 是“*读*”操作可被执行的上限率。例如，让 **/dev/sda** 设备运行“*读*”操作的最大速率是 10 MBps，请运行：

```
~]# echo "8:0 10485760" >
/cgroup/blkio/test/blkio.throttle.read_bps_device
```

blkio.throttle.read_iops_device

此参数用于设定设备执行“*读*”操作次数的上限。“*读*”的操作率以每秒的操作次数来表示。条目有三个字段：*major*、*minor* 和 *operations_per_second*。*major* 和 *minor* 是〈*Linux 分配的设备*〉指定的设备类型和节点数。*operations_per_second* 是“*读*”可被执行的上限率。例如：如要设定 **/dev/sda** 设备执行“*读*”的最大比率为 10 次/秒，请运行：

```
~]# echo "8:0 10" >
/cgroup/blkio/test/blkio.throttle.read_iops_device
```

blkio.throttle.write_bps_device

此参数用于设定设备执行“*写*”操作次数的上限。“*写*”的操作率用“字节/秒”来表示。条目有三个字段：*major*、*minor* 和 *bytes_per_second*。*major* 和 *minor* 是〈*Linux 分配的设备*〉指定的设备类型和节点数。*bytes_per_second* 是“*写*”操作可被执行的上限率。例如，让 **/dev/sda** 设备执行“*写*”操作的最大比率为 10 MBps，请运行：

```
~]# echo "8:0 10485760" >
/cgroup/blkio/test/blkio.throttle.write_bps_device
```

blkio.throttle.write_iops_device

此参数用于设定设备执行 “*写*” 操作次数的上限。“*写*”的操作率以每秒的操作次数来表示。条目有三个字段：*major*、*minor* 和 *operations_per_second*。*major* 和 *minor* 是〈*Linux 分配的设备*〉指定的设备类型和节点数。*operations_per_second* 是“*写*” 操作可被执行的上限率。例如：如要让 **/dev/sda** 设备执行“*写*”操作的最大比率为 10 次/秒，请运行：

```
~]# echo "8:0 10" >
/cgroup/blkio/test/blkio.throttle.write_iops_device
```

blkio.throttle.io_serviced

此参数用于报告 cgroup 根据节流方式在具体设备中执行的 I/O 操作数。条目有四个字段：*major*、*minor*、*operation* 和 *number*。*major* 和 *minor* 是〈*Linux 分配的设备*〉指定的设备类型和节点数，*operation* 代表操作类型（**read**、**write**、**sync** 或者 **async**），*number* 代表操作数。

blkio.throttle.io_service_bytes

此参数用于报告 cgroup 传送到具体设备或者由具体设备中传送出的字节数。***blkio.io_service_bytes*** 和 ***blkio.throttle.io_service_bytes*** 之间的唯一区别是：CFQ 调度程序在请求队列中操作时，前者不会被更新。条目有四个字段：*major*、*minor*、*operation* 和 *bytes*。*major* 和 *minor* 是〈*Linux 分配的设备*〉指定的设备类型和节点数。*operation* 代表操作类型（**read**、**write**、**sync** 或者 **async**）。*bytes* 是被传送的字节数。

A.1.3. blkio 的通用可调参数

以下参数可用于〈第 A.1 节 "blkio"〉所列方法的任意一个。

blkio.reset_stats

此参数用于重设其它伪文件记录的统计数据。请在此文件中写入整数来为 cgroup 重设统计数据。

blkio.time

此参数用于报告 cgroup 对具体设备的 I/O 访问时间。条目有三个字段：*major*、*minor* 和 *time*。*major* 和 *minor* 是〈*Linux 分配的设备*〉指定的设备类型和节点数，*time* 表示时间长度，单位为毫秒（ms）。

blkio.sectors

此参数用于报告 cgroup 转换到具体设备或者由具体设备转换出的扇区数。条目有三个字段：*major*、*minor* 和 *sectors*。*major* 和 *minor* 是〈*Linux 分配的设备*〉指定的设备类型和节点数，*sectors* 是磁盘扇区数。

blkio.avg_queue_size

此参数用于报告：群组存在的整个过程中，cgroup I/O 操作的平均队列大小。每当此 cgroup 队列获得一个 timeslice 时，该队列大小都将被采样。请注意，只有系统设定了 **CONFIG_DEBUG_BLK_CGROUP=y** 后，此报告才可用。

blkio.group_wait_time

此参数用于报告 cgroup 中每一个队列等待 timeslice 的总时间（单位为纳秒：ns）。每当 cgroup 队列获得一个 timeslice 时，此报告就会被更新，因此如果您在 cgroup 等待 timeslice 时读取伪文件，该报告将不会包含当前队列等待操作的时间。请注意，只有系统设定了 **CONFIG_DEBUG_BLK_CGROUP=y** 后，此报告才可用。

blkio.empty_time

此参数用于报告：没有任何等待处理的请求时，cgroup 花费的总时间（单位为纳秒：ns）。每当 cgroup 的列队有等待处理的请求时，报告都会被更新，因此如果您在 cgroup 没有任何等待处理的请求时读取此伪文件，该报告将不会包含消耗在当前空状态中的时间。请注意，只有系统设定了 **CONFIG_DEBUG_BLK_CGROUP=y** 后，此报告才可用。

blkio.idle_time

此参数用于报告：当一个请求比其它队列或者其它群组的请求更好时，调度程序让 cgroup 闲置所消耗的总时间（单位为纳秒：ns）。每当该群组不处于闲置状态时，该报告就会被更新。因此如果您在 cgroup 闲置时读取该伪文件，该报告将不会包含消耗在当前闲置状态的时间。请注意，只有系统设定了 **CONFIG_DEBUG_BLK_CGROUP=y** 后，此报告才可用。

blkio.dequeue

此参数用于报告 cgroup 的 I/O 操作请求被具体设备从队列中移除的次数。条目有三个字段：*major*、*minor* 和 *number*。*major* 和 *minor* 是〈*Linux 分配的设备*〉指定的设备类型和节点数，*number* 是该群组的请求被移除的次数。请注意，只有系统设定了 **`CONFIG_DEBUG_BLK_CGROUP=y`** 后，此报告才可用。

blkio.io_serviced

此参数用于报告 cgroup 根据 CFQ 调度程序在具体设备中执行的 I/O 操作数。条目有四个字段：*major*、*minor*、*operation* 和 *number*。*major* 和 *minor* 是〈*Linux 分配的设备*〉指定的设备类型和节点数，*operation* 代表操作类型（**`read`**、**`write`**、**`sync`** 或者 **`async`**），*number* 代表操作次数。

blkio.io_service_bytes

此参数用于报告 cgroup 根据 CFQ 调度程序转换到具体设备或者由具体设备中转出的字节数。条目有四个字段：*major*、*minor*、*operation* 和 *bytes*。*major* 和 *minor* 是〈*Linux 分配的设备*〉指定的设备类型和节点数，*operation* 代表操作类型（**`read`**、**`write`**、**`sync`** 或者 **`async`**），*bytes* 表示转换的字节数。

blkio.io_service_time

此参数用于报告 cgroup 根据 CFQ 调度程序在具体设备中执行 I/O 操作时，发送请求到完成请求的时间。条目有四个字段：*major*、*minor*、*operation* 和 *time*。*major* 和 *minor* 是〈*Linux 分配的设备*〉指定的设备类型和节点数。*operation* 表示操作类型（**`read`**、**`write`**、**`sync`** 或者 **`async`**），*time* 表示时间长度，单位为纳秒（ns）。此处使用纳秒而不是较大的单位，是为了对固态设备报告也有意义。

blkio.io_wait_time

此参数用于报告 cgroup 在指定设备中执行 I/O 操作时，在调度程序队列中等待服务的总时间。当您解读该报告时，请注意：

- 报告的时间可能比消耗的时间长，因为报告的时间是该 cgroup 所有 I/O 操作的总和，而不是该 cgroup 本身等待 I/O 操作的时间。要查找该群组作为整体的等待时间，请使用 **`blkio.group_wait_time`** 参数。
- 如果设备包含 **`queue_depth`** > 1，则报告只包括向该设备发送请求之前的时间，而不包括该设备将请求重新排序时等待服务的时间。

条目有四个字段：*major*、*minor*、*operation* 和 *time*。*major* 和 *minor* 是〈*Linux 分配的设备*〉指定的设备类型和节点数，*operation* 表示操作类型（**`read`**、**`write`**、**`sync`** 或者 **`async`**），*time* 表示时间长度，单位为纳秒（ns）。此处使用纳秒而不是较大的单位，是为了对固态设备报告也有意义。

blkio.io_merged

此参数用于报告 cgroup 将 BIOS 请求合并到 I/O 操作请求的次数。条目有两个字段：*number* 和 *operation*。*number* 是请求次数，*operation* 表示操作类型（**`read`**、**`write`**、**`sync`** 或者 **`async`**）。

blkio.io_queued

此参数用于报告 cgroup 排队请求 I/O 操作的次数。条目有两个字段：*number* 和 *operation*。*number* 是请求次数，*operation* 表示操作类型（**`read`**、**`write`**、**`sync`** 或者 **`async`**）。

A.1.4. 示例应用

如想简单测试在两个不同 cgroup 中，使用不同 **`blkio.weight`** 值运行两个 **`dd`** 线程，请参阅例 A.1 “blkio 权

重分配”。

例 A.1. blkio 权重分配

1. 挂载 **blkio** 子系统：

```
~]# mount -t cgroup -o blkio blkio /cgroup/blkio/
```

2. 为 **blkio** 子系统创建两个 cgroup：

```
~]# mkdir /cgroup/blkio/test1/
~]# mkdir /cgroup/blkio/test2/
```

3. 在之前创建的 cgroup 设定不同 **blkio** 权重：

```
~]# echo 1000 > /cgroup/blkio/test1/blkio.weight
~]# echo 500 > /cgroup/blkio/test2/blkio.weight
```

4. 创建两个大文件：

```
~]# dd if=/dev/zero of=file_1 bs=1M count=4000
~]# dd if=/dev/zero of=file_2 bs=1M count=4000
```

以上指令所建的文件大小是 4 GB（**file_1** 和 **file_2**）。

5. 对每个测试 cgroup ，在一个大文件中执行 **dd** 指令（该指令可以读取文件内容，并将其输出到空设备）：

```
~]# cgexec -g blkio:test1 time dd if=file_1 of=/dev/null
~]# cgexec -g blkio:test2 time dd if=file_2 of=/dev/null
```

两个指令在结束后都会输出完成时间。

6. 您可以使用 **iotop** 实用工具实时监控两个同时运行的 **dd** 线程。要安装 **iotop** 实用工具，请以 root 身份执行 **yum install iotop** 指令。运行之前启动的 **dd** 线程时，您可以在 **iotop** 实用工具中看到以下结果示例：

```
Total DISK READ: 83.16 M/s | Total DISK WRITE: 0.00 B/s
    TIME  TID  PRIO  USER     DISK READ  DISK WRITE  SWAPIN      IO
COMMAND
15:18:04 15071 be/4 root       27.64 M/s    0.00 B/s  0.00 % 92.30 %
dd if=file_2 of=/dev/null
15:18:04 15069 be/4 root       55.52 M/s    0.00 B/s  0.00 % 88.48 %
dd if=file_1 of=/dev/null
```

为获得例 A.1 “blkio 权重分配”最准确的结果，请优先执行 **dd** 指令、清除所有文件系统缓存并使用下列指令释放缓存页、目录项和索引节点：

```
~]# sync
~]# echo 3 > /proc/sys/vm/drop_caches
```

您可以启用“*群组隔离*”，它能使用吞吐量来提供更稳定的群组隔离。当群组隔离被禁用，只有工作量是顺序时，公平才能实现。默认情况下，群组隔离是启用状态，这样即便 I/O 工作量随机也可以保证公平。如要启用群组隔离，请执行下列指令：

```
~]# echo 1 > /sys/block/<disk_device>/queue/iosched/group_isolation
```

其中，*<disk_device>* 代表所需设备名称，例如：**sda**。

A.2. cpu

cpu 子系统可以调度 cgroup 对 CPU 的获取量。可用以下两个调度程序来管理对 CPU 资源的获取：

- *完全公平调度程序（CFS）* — 一个比例分配调度程序，可根据任务优先级／权重或 cgroup 分得的份额，在任务群组（cgroups）间按比例分配 CPU 时间（CPU 带宽）。关于如何使用 CFS 进行资源控制，请参阅〈第 A.2.1 节 “CFS 可调度参数”〉。
- *实时调度程序（RT）* — 一个任务调度程序，可对实时任务使用 CPU 的时间进行限定。关于如何进行限定，请参阅〈第 A.2.2 节 “RT 可调参数”〉。

A.2.1. CFS 可调度参数

在 CFS 中，如果系统有足够的空闲 CPU 周期，那么 cgroup 可获得比其自有份额更多的 CPU 可用量，因为该调度程序有连续工作的特性。此情况通常会在 cgroup 根据相关共享消耗 CPU 时间时出现。在需要对 cgroup 的 CPU 可用量做出硬性限制时（即任务的 CPU 时间不能超过一个特定量），可使用强制上限。

以下选项可用来配置强制上限或者 CPU 相对份额：

强制上限的可调参数

cpu.cfs_period_us

此参数可以设定重新分配 cgroup 可用 CPU 资源的时间间隔，单位为微秒（µs，这里以 “***us***” 表示）。如果一个 cgroup 中的任务在每 1 秒钟内有 0.2 秒的时间可存取一个单独的 CPU，则请将 **cpu.rt_runtime_us** 设定为 **2000000**，并将 **cpu.rt_period_us** 设定为 **1000000**。**cpu.cfs_quota_us** 参数的上限为 1 秒，下限为 1000 微秒。

cpu.cfs_quota_us

此参数可以设定在某一阶段（由 **cpu.cfs_period_us** 规定）某个 cgroup 中所有任务可运行的时间总量，单位为微秒（µs，这里以 "***us***" 代表）。一旦 cgroup 中任务用完按配额分得的时间，它们就会被在此阶段的时间提醒限制流量，并在进入下阶段前禁止运行。如果 cgroup 中任务在每 1 秒内有 0.2 秒，可对单独 CPU 进行存取，请将 **cpu.cfs_quota_us** 设定为 **200000**，**cpu.cfs_period_us** 设定为 **1000000**。请注意，配额和时间段参数都根据 CPU 来操作。例如，如要让一个进程完全利用两个 CPU，请将 **cpu.cfs_quota_us** 设定为 **200000**，**cpu.cfs_period_us** 设定为 **100000**。

如将 **cpu.cfs_quota_us** 的值设定为 **-1**，这表示 cgroup 不需要遵循任何 CPU 时间限制。这也是每个 cgroup 的默认值（root cgroup 除外）。

cpu.stat

此参数通过以下值报告 CPU 时间统计：

- **nr_periods** — 经过的周期间隔数（如 **cpu.cfs_period_us** 中所述）。

- **nr_throttled** — cgroup 中任务被节流的次数（即耗尽所有按配额分得的可用时间后，被禁止运行）。
- **throttled_time** — cgroup 中任务被节流的时间总计（以纳秒为单位）。

相对比例的可调参数

cpu.shares

此参数用一个整数来设定 cgroup 中任务 CPU 可用时间的相对比例。例如： **cpu.shares** 设定为 **100** 的任务，即便在两个 cgroup 中，也将获得相同的 CPU 时间；但是 **cpu.shares** 设定为 **200** 的任务与 **cpu.shares** 设定为 **100** 的任务相比，前者可使用的 CPU 时间是后者的两倍，即便它们在同一个 cgroup 中。**cpu.shares** 文件设定的值必须大于等于 **2**。

请注意：在多核系统中，CPU 时间比例是在所有 CPU 核中分配的。即使在一个多核系统中，某个 cgroup 受限制而不能 100% 使用 CPU，它仍可以 100% 使用每个单独的 CPU 核。请参考以下示例：如果 cgroup **A** 可使用 CPU 的 25%，cgroup **B** 可使用 CPU 的 75%，在四核系统中启动四个消耗大量 CPU 的进程（一个在 **A** 中，三个在 **B** 中）后，会得到以下 CPU 分配结果：

表 A.1. CPU 分配比例

PID	cgroup	CPU	CPU 共享
100	A	0	CPU0 的 100%
101	B	1	CPU1 的 100%
102	B	2	CPU2 的 100%
103	B	3	CPU3 的 100%

使用 “相对比例” 设定 CPU 获取量来管理资源时，有两个问题需要注意：

- 因为 CFS 不需要相同的 CPU 使用量，所以很难预测 cgroup 的 CPU 可用时间。当一个 cgroup 中的任务处于闲置状态且不使用任何 CPU 时间时，剩余的时间会被收集到未使用的 CPU 循环全局池中。其它 cgroup 可以从这个池中借用 CPU 循环。
- cgroup 实际可用的 CPU 时间会根据该系统中 cgroup 的数量有所不同。如果一个 cgroup 的相对份额是 **1000**，另外两个 cgroup 的相对份额是 **500**，在所有 cgroup 中的进程都尝试使用 100% CPU 时间的条件下，第一个 cgroup 将有 50% 的 CPU 时间。但如果新添加一个相对份额为 **1000** 的 cgroup，则第一个 cgroup 只允许使用 33% 的 CPU 时间（剩余的 cgroup 则使用 16.5%、16.5% 和 33% 的 CPU 时间）。

A.2.2. RT 可调参数

RT 调度程序与 CFS 的强制上限（详情请参阅〈第 A.2.1 节 “CFS 可调度参数”〉）类似，但只限制实时任务对 CPU 的存取。一个实时任务存取 CPU 的时间可以通过为每个 cgroup 分配运行时间和时段来设定。然后，在运行时间的特定时间段，cgroup 中的所有任务会被允许存取 CPU（例如：可允许 cgroup 中的任务每秒中运行 0.1 秒）。

cpu.rt_period_us

此参数可以设定在某个时间段中 ，每隔多久，cgroup 对 CPU 资源的存取就要重新分配，单位为微秒（µs，这里以“*us*”表示），只可用于实时调度任务。如果某个 cgroup 中的任务，每秒内有 0.2 秒可存取 CPU 资源，则请将 **cpu.rt_runtime_us** 设定为 **200000**，并将 **cpu.rt_period_us** 设定为 **1000000**。

cpu.rt_runtime_us

此参数可以指定在某个时间段中， cgroup 中的任务对 CPU 资源的最长连续访问时间，单位为微秒

（µs，这里以“*us*”表示），只可用于实时调度任务。建立这个限制是为了防止一个 cgroup 中的任务独占 CPU 时间。如果 cgroup 中的任务，在每秒内有 0.2 秒可存取 CPU 资源，请将 **cpu.rt_runtime_us** 设定为 **200000**，并将 **cpu.rt_period_us** 设定为 **1000000**。请注意：运行时间和阶段参数会根据 CPU 操作。例如：如要允许一个实时任务完全利用两个 CPU，请将 **cpu.cfs_quota_us** 设定为 **200000** 并将 **cpu.cfs_period_us** 设定为 **100000**。

A.2.3. 示例应用

例 A.2. 限制 CPU 存取

以下示例假设您已配置 cgroup 层级并且**cpu** 子系统已挂载到您的系统中：

- 如要让一个 cgroup 使用一个 CPU 的 25%，同时另一个 cgroup 使用此 CPU 的 75%，请使用以下指令：

```
~]# echo 250 > /cgroup/cpu/blue/cpu.shares
~]# echo 750 > /cgroup/cpu/red/cpu.shares
```

- 如要让一个 cgroup 完全使用一个 CPU，请使用以下指令：

```
~]# echo 10000 > /cgroup/cpu/red/cpu.cfs_quota_us
~]# echo 10000 > /cgroup/cpu/red/cpu.cfs_period_us
```

- 如要让一个 cgroup 使用 CPU 的 10%，请使用以下指令：

```
~]# echo 10000 > /cgroup/cpu/red/cpu.cfs_quota_us
~]# echo 100000 > /cgroup/cpu/red/cpu.cfs_period_us
```

- 在多核系统中，如要让一个 cgroup 完全使用两个 CPU 核，请使用以下指令：

```
~]# echo 200000 > /cgroup/cpu/red/cpu.cfs_quota_us
~]# echo 100000 > /cgroup/cpu/red/cpu.cfs_period_us
```

A.3. cpuacct

CPU 统计（CPU accounting）（**cpuacct**）子系统会自动生成报告来显示 cgroup 任务所使用的 CPU 资源，其中包括子群组任务。报告有三种：

cpuacct.usage

报告此 cgroup 中所有任务（包括层级中的低端任务）使用 CPU 的总时间（纳秒）。

注意

如要重新设定 **cpuacct.usage** 的值，请执行以下指令：

```
~]# echo 0 > /cgroup/cpuacct/cpuacct.usage
```

上述指令也将重置 **cpuacct.usage_percpu** 中的值。

cpuacct.stat

报告此 cgroup 的所有任务（包括层级中的低端任务）使用的用户和系统 CPU 时间，方式如下：

- **user** — 用户模式中任务使用的 CPU 时间。
- **system** — 系统（kernel）模式中任务使用的 CPU 时间。

CPU 时间将报告于 **USER_HZ** 变量定义的单位中。

cpuacct.usage_percpu

报告 cgroup 中所有任务（包括层级中的低端任务）在每个 CPU 中使用的 CPU 时间（纳秒）。

A.4. cpuset

cpuset 子系统可以为 cgroup 分配独立 CPU 和内存节点。可根据以下参数来设定 cpuset，每个参数都在 cgroup 虚拟文件系统的一个单独 “*伪文件*” 里：

重要

一些子系统有强制参数，在您将任务移至使用这些子系统的 cgroup 中之前，这些参数必须被设定。 例如，一个使用 **cpuset** 子系统的 cgroup, 在您将任务移至此 cgroup 前，**cpuset.cpus** 和 **cpuset.mems** 参数必须被设定。

cpuset.cpus（强制）

设定该 cgroup 任务可以访问的 CPU。这是一个逗号分隔列表，格式为 ASCII，小横线（"**-**"）代表范围。例如：

```
0-2,16
```

表示 CPU 0、1、2 和 16。

cpuset.mems（强制）

设定该 cgroup 中任务可以访问的内存节点。这是一个逗号分隔列表，格式为 ASCII，小横线（"**-**"）代表范围。例如：

```
0-2,16
```

表示内存节点 0、1、2 和 16。

cpuset.memory_migrate

包含一个标签（**0** 或者 **1**），用来指定当 **cpuset.mems** 的值更改时，是否应该将内存中的页迁移到新节点。默认情况下禁止内存迁移（**0**）且页就保留在原来分配的节点中，即使此节点不再是 **cpuset.mems** 指定的节点。如果启用（**1**），系统会将页迁移到 **cpuset.mems** 指定的新参数的内存节点中，如果可能的话会保留其相对位置。例如：如果页位于 **cpuset.mems** 指定列表的第二个节点中，现在页将会重新分配到 **cpuset.mems** 指定列表的第二个节点中，如果这个位置是可用的。

cpuset.cpu_exclusive

包含标签（**0** 或者 **1**），它可以指定：其它 cpuset 及其父、子 cpuset 是否可共享该 cpuset 的特定 CPU。默认情况下（**0**），CPU 不会专门分配给某个 cpuset 。

cpuset.mem_exclusive

包含标签（**0** 或者 **1**），它可以指定：其它 cpuset 是否可共享该 cpuset 的特定内存节点。默认情况下（**0**），内存节点不会专门分配给某个 cpuset 。为某个 cpuset 保留其专用内存节点（**1**）与使用 **cpuset.mem_hardwall** 参数启用内存 hardwall 功能是一样的。

cpuset.mem_hardwall

包含标签（**0** 或者 **1**），它可以指定：内存页和缓冲数据的 kernel 分配是否受到 cpuset 特定内存节点的限制。默认情况下 **0**，页面和缓冲数据在多用户进程间共享。启用 hardwall 时（**1**）每个任务的用户分配可以保持独立。

cpuset.memory_pressure

一份只读文件，包含该 cpuset 进程生成的“*内存压力*”运行平均。启用 **cpuset.memory_pressure_enabled** 时，该伪文件中的值会自动更新，除非伪文件包含 **0** 值。

cpuset.memory_pressure_enabled

包含标签（**0** 或者 **1**），它可以设定系统是否计算该 cgroup 进程生成的“*内存压力*”。计算出的值会输出到 **cpuset.memory_pressure**，代表进程试图释放被占用内存的速率，报告值为：每秒尝试回收内存的整数值再乘以 1000。

cpuset.memory_spread_page

包含标签（**0** 或者 **1**），它可以设定文件系统缓冲是否应在该 cpuset 的内存节点中均匀分布。默认情况下 **0**，系统不会为这些缓冲平均分配内存页面，缓冲被置于生成缓冲的进程所运行的同一节点中。

cpuset.memory_spread_slab

包含标签（**0** 或者 **1**），它可以设定是否在 cpuset 间平均分配用于文件输入 / 输出操作的 kernel 高速缓存板。默认情况下 **0**，kernel 高速缓存板不被平均分配，高速缓存板被置于生成它们的进程所运行的同一节点中。

cpuset.sched_load_balance

包含标签（**0** 或者 **1**），它可以设定 kernel 是否在该 cpuset 的 CPU 中平衡负载。默认情况下 **1**，kernel 将超载 CPU 中的进程移动到负载较低的 CPU 中以便平衡负载。

请注意：如果任意一个父 cgroup 启用负载平衡，那么在 cgroup 中设定这个标签将没有任何效果，因为负载平衡已在更高层级中运行。因此，要禁用 cgroup 中的负载平衡，则层级中的每一个父 cgroup 负载平衡都要禁用。这里您还应该考虑是否在所有平级 cgroup 中启用负载平衡。

cpuset.sched_relax_domain_level

包含 **`-1`** 到一个小正数间的整数，它代表 kernel 应尝试平衡负载的 CPU 宽度范围。如果禁用 **`cpuset.sched_load_balance`**，则该值无意义。

根据不同系统构架，这个值的具体效果不同，但以下值是常用的：

cpuset.sched_relax_domain_level 值

值	效果
`-1`	平衡负载的系统默认值
`0`	不执行直接负载平衡；负载平衡只是阶段性的
`1`	对同一核中的线程进行直接负载平衡
`2`	对同一软件包中的线程进行直接负载平衡
`3`	对同一节点或者扇叶中的线程进行直接负载平衡
`4`	对不使用统一内存访问（NUMA）构架中的多个 CPU 进行直接负载平衡
`5`	对使用统一内存访问（NUMA）构架中的多个 CPU 进行直接负载平衡

A.5. devices

`devices` 子系统允许或者拒绝 cgroup 任务存取设备。

重要

设备白名单（Device Whitelist）（**`devices`**）子系统被视为 Red Hat Enterprise Linux 7 的“技术预览”。

Red Hat Enterprise Linux 7 订阅服务暂不支持 *“技术预览”* 功能，可能是由于功能还不完备，所以通常不适合生产使用。但 Red Hat 在操作系统中包含这些功能，为了方便用户并提供更多功能。您会发现这些功能可能在非生产环境中很有用，也请您提供反馈意见和功能建议，以便今后全面支持“技术预览”。

devices.allow

指定 cgroup 任务可访问的设备。每个条目有四个字段：*type*、*major*、*minor* 和 *access*。*type*、*major* 和 *minor* 字段使用的值对应〈*Linux 分配的设备*〉（也称为〈*Linux 设备列表*〉）指定的设备类型和节点数，如需此介绍，请访问 http://www.kernel.org/doc/Documentation/devices.txt。

type

type 的值有三种：

- **a** —— 可用于所有设备，*“字符设备”*和*“块设备”* 均可
- **b** —— 指定一个块设备
- **c** —— 指定一个字符设备

major, minor

major 和 *minor* 是〈*Linux 分配的设备*〉指定的设备节点数。major 数和 minor 数用冒号隔

major 和 *minor* 是（*Linux* 分配的设备）指定的设备节点数。major 数和 minor 数用冒号隔开。例如：**8** 是指定 SCSI 磁盘驱动器的 major 数；**1** 是指定第一个 SCSI 磁盘驱动器中第一个分区的 minor 数；因此 **8:1** 完整地指定该分区，与 **/dev/sda1** 的一个文件系统位置对应。

***** 可代表所有主要或所有次要的设备节点，例如：**9:***（所有 RAID 设备）或者 ***:***（所有设备）。

access

access 是以下一个或者多个字母的序列：

- **r** —— 允许任务从指定设备中读取
- **w** —— 允许任务对指定设备写入
- **m** —— 允许任务创建还不存在的设备文件

例如：当将 *access* 被指定为 **r** 时，则任务只能从指定设备中读取，但将 *access* 指定为 **rw** 时，则任务既可从该设备中读取，也可向该设备写入。

devices.deny

指定 cgroup 任务无权访问的设备。条目语法与 **devices.allow** 一致。

devices.list

报告 cgroup 任务对其访问受限的设备。

A.6. freezer

freezer 子系统可以暂停或者恢复 cgroup 中的任务。

freezer.state

freezer.state 只能用于非 root cgroup 中，它有三个可能的值：

- **FROZEN** —— cgroup 中的任务已被暂停。
- **FREEZING** —— 系统正在暂停 cgroup 中的任务。
- **THAWED** —— cgroup 中的任务已恢复。

如要暂停某一进程：

1. 请将该进程移动到已附加 **freezer** 子系统层级的 cgroup 中。
2. 冻结此 cgroup 以便暂停其中的进程。

进程不能移至已暂停（已冻结）的 cgroup 中。

请注意：虽然 **FROZEN** 和 **THAWED** 值可写入 **freezer.state**，但 **FREEZING** 无法被写入，只能读取。

A.7. memory

memory 子系统自动生成 cgroup 任务使用内存资源的报告，并限定这些任务所用内存的大小：

memory.stat

报告大范围内存统计，见下表：

表 A.2. memory.stat 报告的值

统计数据	描述
cache	缓存页，包括 **tmpfs**（**shmem**），单位为字节
rss	匿名和 swap 缓存，“*不*”包括 **tmpfs**（**shmem**），单位为字节
mapped_file	memory-mapped 映射文件大小，包括 **tmpfs**（**shmem**），单位为字节
pgpgin	读入内存的页数
pgpgout	从内存中读出的页数
swap	swap 用量，单位为字节
active_anon	激活的“近期最少使用”（least-recently-used，LRU）列表中的匿名和 swap 缓存，包括 **tmpfs**（**shmem**），单位为字节
inactive_anon	未激活的 LRU 列表中的匿名和 swap 缓存，包括 **tmpfs**（**shmem**），单位为字节
active_file	激活的 LRU 列表中的 file-backed 内存，以字节为单位
inactive_file	未激活 LRU 列表中的 file-backed 内存，以字节为单位
unevictable	无法收回的内存，以字节为单位
hierarchical_memory_limit	包含 **memory** cgroup 层级的内存限制，单位为字节
hierarchical_memsw_limit	包含 **memory** cgroup 层级的内存加 swap 限制，单位为字节

另外，这些文件除 **hierarchical_memory_limit** 和 **hierarchical_memsw_limit** 之外，都有一个对应前缀 **total**，它不仅可在该 cgroup 中报告，还可在其子 cgroup 中报告。例如：**swap** 报告 cgroup 的 swap 用量，**total_swap** 报告该 cgroup 及其所有子群组的 swap 用量总和。

当您解读 **memory.stat** 报告的数值时，请注意各个统计数据之间的关系：

- **active_anon** + **inactive_anon** = 匿名内存 + **tmpfs** 文件缓存 + swap 缓存

  因此，**active_anon** + **inactive_anon** ≠ **rss**，因为 **rss** 不包括 **tmpfs**。

- **active_file** + **inactive_file** = 缓存 - **tmpfs** 大小

memory.usage_in_bytes

报告 cgroup 中进程当前所用的内存总量（以字节为单位）。

memory.memsw.usage_in_bytes

报告该 cgroup 中进程当前所用的内存量和 swap 空间总和（以字节为单位）。

memory.max_usage_in_bytes

报告 cgroup 中进程所用的最大内存量（以字节为单位）。

memory.memsw.max_usage_in_bytes

报告该 cgroup 中进程的最大内存用量和最大 swap 空间用量（以字节为单位）。

memory.limit_in_bytes

设定用户内存（包括文件缓存）的最大用量。如果没有指定单位，则该数值将被解读为字节。但是可以使用后缀代表更大的单位 —— **k** 或者 **K** 代表千字节，**m** 或者 **M** 代表兆字节 ，**g** 或者 **G** 代表千

兆字节。

您不能使用 **memory.limit_in_bytes** 限制 root cgroup；您只能对层级中较低的群组应用这些值。

在 **memory.limit_in_bytes** 中写入 **-1** 可以移除全部已有限制。

memory.memsw.limit_in_bytes

设定内存与 swap 用量之和的最大值。如果没有指定单位，则该值将被解读为字节。但是可以使用后缀代表更大的单位 —— **k** 或者 **K** 代表千字节，**m** 或者 **M** 代表兆字节，**g** 或者 **G** 代表千兆字节。

您不能使用 **memory.memsw.limit_in_bytes** 来限制 root cgroup；您只能对层级中较低的群组应用这些值。

在 **memory.memsw.limit_in_bytes** 中写入 **-1** 可以删除已有限制。

重要

在设定 **memory.memsw.limit_in_bytes** 参数*"之前"*设定 **memory.limit_in_bytes** 参数非常重要：顺序颠倒会导致错误。这是因为 **memory.memsw.limit_in_bytes** 只有在消耗完所有内存限额（之前在 **memory.limit_in_bytes** 中设定）后方可用。

请参考下列例子：为某一 cgroup 设定 **memory.limit_in_bytes = 2G** 和 **memory.memsw.limit_in_bytes = 4G**， 可以让该 cgroup 中的进程分得 2GB 内存，并且一旦用尽，只能再分得 2GB swap。**memory.memsw.limit_in_bytes** 参数表示内存和 swap 的总和。没有设置 **memory.memsw.limit_in_bytes** 参数的 cgroup 的进程可以使用全部可用 swap （当限定的内存用尽后），并会因为缺少可用 swap 触发 Out of Memory（内存不足） 状态。

/etc/cgconfig.conf 文件中 **memory.limit_in_bytes** 和 **memory.memsw.limit_in_bytes** 参数的顺序也很重要。以下是正确的配置示例：

```
memory {
    memory.limit_in_bytes = 1G;
    memory.memsw.limit_in_bytes = 1G;
}
```

memory.failcnt

报告内存达到 **memory.limit_in_bytes** 设定的限制值的次数。

memory.memsw.failcnt

报告内存和 swap 空间总和达到 **memory.memsw.limit_in_bytes** 设定的限制值的次数。

memory.force_empty

当设定为 **0** 时，该 cgroup 中任务所用的所有页面内存都将被清空。这个接口只可在 cgroup 没有任务时使用。如果无法清空内存，请在可能的情况下将其移动到父 cgroup 中。移除 cgroup 前请使用 **memory.force_empty** 参数以免将废弃的页面缓存移动到它的父 cgroup 中。

memory.swappiness

将 kernel 倾向设定为换出该 cgroup 中任务所使用的进程内存，而不是从页高速缓冲中再生页面。

这与 **/proc/sys/vm/swappiness** 为整体系统设定的倾向、计算方法相同。默认值为 **60**。低于 **60** 会降低 kernel 换出进程内存的倾向；高于 **0** 会增加 kernel 换出进程内存的倾向。高于 **100** 时，kernel 将开始换出作为该 cgroup 中进程地址空间一部分的页面。

请注意：值 **0** 不会阻止进程内存被换出；系统内存不足时，换出仍可能发生，因为全局虚拟内存管理逻辑不读取该 cgroup 值。要完全锁定页面，请使用 **mlock()** 而不是 cgroup。

您不能更改以下群组的 swappiness：

- root cgroup，它使用 **/proc/sys/vm/swappiness** 设定的 swappiness。
- 有子群组的 cgroup。

memory.use_hierarchy

包含标签（**0** 或者 **1**），它可以设定是否将内存用量计入 cgroup 层级的吞吐量中。如果启用（**1**），内存子系统会从超过其内存限制的子进程中再生内存。默认情况下（**0**），子系统不从任务的子进程中再生内存。

memory.oom_control

包含标签（**0** 或者 **1**），它可以为 cgroup 启用或者禁用“内存不足”（Out of Memory，OOM） 终止程序。如果启用（**0**），尝试消耗超过其允许内存的任务会被 OOM 终止程序立即终止。默认情况下，所有使用 **memory** 子系统的 cgroup 都会启用 OOM 终止程序。要禁用它，请在 **memory.oom_control** 文件中写入 **1**：

```
~]# echo 1 > /cgroup/memory/lab1/memory.oom_control
```

禁用 OOM 杀手程序后，尝试使用超过其允许内存的任务会被暂停，直到有额外内存可用。

memory.oom_control 文件也在 **under_oom** 条目下报告当前 cgroup 的 OOM 状态。如果该 cgroup 缺少内存，则会暂停它里面的任务。**under_oom** 条目报告值为 **1**。

memory.oom_control 文件可以使用 API 通知来报告 OOM 情况的出现。 。

A.7.1. 示例应用

例 A.3. OOM 控制和通知

以下示例将演示当 cgroup 中任务尝试使用超过其允许的内存时， OOM 终止程序的工作过程，以及通知处理程序是如何报告 OOM 状态的：

1. 在层级中附加 **memory** 子系统，并创建一个 cgroup：

```
~]# mount -t memory -o memory memory /cgroup/memory
~]# mkdir /cgroup/memory/blue
```

2. 将 **blue** cgroup 中任务可用的内存量设定为 100MB：

```
~]# echo 104857600 > memory.limit_in_bytes
```

3. 进入 **blue** 目录并确定已启用 OOM 终止程序：

```
~]# cd /cgroup/memory/blue
blue]# cat memory.oom_control
```

```
oom_kill_disable 0
under_oom 0
```

4. 将当前 shell 进程移动到 **blue** cgroup 的 **tasks** 文件中，以便在这个 shell 中启动的其它所有进程会自动移至 **blue** cgroup：

```
blue]# echo $$ > tasks
```

5. 启动测试程序，尝试分配超过您在步骤 2 中设定限额的内存量。**blue** cgroup 消耗完内存后，OOM 终止程序会终止测试程序，并在标准输出中报告 **Killed**：

```
blue]# ~/mem-hog
Killed
```

以下是测试程序实例 [1]：

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

#define KB (1024)
#define MB (1024 * KB)
#define GB (1024 * MB)

int main(int argc, char *argv[])
{
 char *p;

again:
 while ((p = (char *)malloc(GB)))
  memset(p, 0, GB);

 while ((p = (char *)malloc(MB)))
  memset(p, 0, MB);

 while ((p = (char *)malloc(KB)))
  memset(p, 0,
    KB);

 sleep(1);

 goto again;

 return 0;
}
```

6. 禁用 OOM 杀手程序，然后重新运行测试程序。这次该测试程序会暂停并等待额外的内存释放：

```
blue]# echo 1 > memory.oom_control
blue]# ~/mem-hog
```

7. 虽然测试程序处于暂停状态，但请注意该 cgroup 的 **under_oom** 状态已更改，表示该 cgroup 缺少可用内存：

```
~]# cat /cgroup/memory/blue/memory.oom_control
oom_kill_disable 1
under_oom 1
```

重启 OOM 终止程序可以立即终止该测试程序。

8. 如要收到关于每一个 OOM 的通知，请创建一个 指定的程序。 例如 [2]：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/eventfd.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

static inline void die(const char *msg)
{
 fprintf(stderr, "error: %s: %s(%d)\n", msg, strerror(errno),
errno);
 exit(EXIT_FAILURE);
}

static inline void usage(void)
{
 fprintf(stderr, "usage: oom_eventfd_test <cgroup.event_control>
<memory.oom_control>\n");
 exit(EXIT_FAILURE);
}

#define BUFSIZE 256

int main(int argc, char *argv[])
{
 char buf[BUFSIZE];
 int efd, cfd, ofd, rb, wb;
 uint64_t u;

 if (argc != 3)
  usage();

 if ((efd = eventfd(0, 0)) == -1)
  die("eventfd");

 if ((cfd = open(argv[1], O_WRONLY)) == -1)
  die("cgroup.event_control");

 if ((ofd = open(argv[2], O_RDONLY)) == -1)
  die("memory.oom_control");

 if ((wb = snprintf(buf, BUFSIZE, "%d %d", efd, ofd)) >= BUFSIZE)
  die("buffer too small");

 if (write(cfd, buf, wb) == -1)
```

```
  die("write cgroup.event_control");

 if (close(cfd) == -1)
  die("close cgroup.event_control");

 for (;;) {
  if (read(efd, &u, sizeof(uint64_t)) != sizeof(uint64_t))
   die("read eventfd");

  printf("mem_cgroup oom event received\n");
 }

 return 0;
}
```

上述程序会探测 OOM 状态（从被命令列指定为参数的 cgroup 中），并使用 **mem_cgroup oom event received** 字符串在标准输出中报告。

9. 在一个独立的控制台中运行上述通知处理程序，并将 **blue** cgroup 的控制文件指定为参数：

```
~]$ ./oom_notification /cgroup/memory/blue/cgroup.event_control
/cgroup/memory/blue/memory.oom_control
```

10. 在另一个控制台中运行 **mem_hog** 测试程序，以便生成 OOM 状态，并查看 **oom_notification** 程序在标准输出中的报告：

```
blue]# ~/mem-hog
```

A.8. net_cls

net_cls 子系统使用等级识别符（classid）标记网络数据包，这让 Linux 流量管控器（**tc**）可以识别从特定 cgroup 中生成的数据包。可配置流量管控器，让其为不同 cgroup 中的数据包设定不同的优先级。

net_cls.classid

net_cls.classid 包含表示流量控制 *handle* 的单一数值。从 **net_cls.classid** 文件中读取的 ***classid*** 值是十进制格式，但写入该文件的值则为十六进制格式。例如：**0x100001** 表示控制点通常写为 *iproute2* 所用的 **10:1** 格式。在 **net_cls.classid** 文件中，将以数字 **1048577** 表示。

这些控制点的格式为：**0x*AAAABBBB***，其中 *AAAA* 是十六进制主设备号，*BBBB* 是十六进制副设备号。您可以忽略前面的零；**0x10001** 与 **0x00010001** 一样，代表 **1:1**。以下是在 **net_cls.classid** 文件中设定 **10:1** 控制点的示例：

```
~]# echo 0x100001 > /cgroup/net_cls/red/net_cls.classid
~]# cat /cgroup/net_cls/red/net_cls.classid
1048577
```

请参考 **tc** 的 manual page 来了解如何配置流量管控器来使用 **net_cls** 添加到网络数据包中的控制点。

A.9. net_prio

网络优先权（**net_prio**）子系统可以为各个 cgroup 中的应用程序动态配置每个网络接口的流量优先级。网络优先级是一个分配给网络流量的数值，可在系统内部和网络设备间使用。网络优先级用来区分发送、排队以及丢失的数据包。可使用 **tc** 命令设定网络优先级（使用 **tc** 命令设定网络优先权的方法将不在本指南中介绍，详情请参考 **tc** manual page）。

通常程序会使用 **SO_PRIORITY** 插槽选项设定其流量的优先级。但应用程序常常没有设定优先级数值的代码，或者程序的流量是特定的，并且不提供确定的优先级。

在 cgroup 中使用 **net_prio** 子系统可让管理员将某个进程分配给具体的 cgroup，该 cgroup 会为指定网络接口的传出流量定义优先级。

net_prio.prioidx

只读文件。它包含一个特有整数值，kernel 使用该整数值作为这个 cgroup 的内部代表。

net_prio.ifpriomap

包含优先级图谱，这些优先级被分配给源于此群组进程的流量以及通过不同接口离开系统的流量。该图用 ***<network_interface> <priority>*** 的形式以成对列表表示：

```
~]# cat /cgroup/net_prio/iscsi/net_prio.ifpriomap
eth0 5
eth1 4
eth2 6
```

net_prio.ifpriomap 文件的目录可以使用上述格式，通过将字符串回显至文件的方式来修改。例如：

```
~]# echo "eth0 5" > /cgroup/net_prio/iscsi/net_prio.ifpriomap
```

上述指令将强制设定任何源于 **iscsi** **net_prio** cgroup 进程的流量和 **eth0** 网络接口传出的流量的优先级为 **5**。父 cgroup 也有可写入的 **net_prio.ifpriomap** 文件，可以设定系统默认优先级。

A.10. ns

ns 子系统提供了一个将进程分组到不同 *namespace* 的方法。在一个名称空间中，进程可彼此互动，但与在其它名称空间中运行的进程隔绝。当这些单独的名称空间被用于操作系统级别的虚拟化时，有时也被称为*"容器"*（container）。

A.11. perf_event

当 **perf_event** 子系统被附加到层级时，该层级中所有 cgroup 都可以用来将进程和线程分组，之后可以使用 **perf** 工具对这些进程和线程监控，这与监控单独进程、独立线程或者单个 CPU 相反。使用 **perf_event** 子系统的 cgroup 不包含任何特殊可调参数，除了〈第 A.12 节 "常用可调参数"〉列出的常用参数。

关于如何使用 **perf** 工具来监控 cgroup 任务，请参阅《Red Hat Enterprise Linux *开发者指南*》，具体请访问 http://access.redhat.com/knowledge/docs/Red_Hat_Enterprise_Linux/ 。

A.12. 常用可调参数

无论 cgroup 使用哪个子系统，下列参数都将出现在每一个被创建的 cgroup 中。

tasks

包含一系列在 cgroup 中运行的进程（由它们的 PID 表示）。PID 列表不一定是有序的，也不一定是特有的（也就是说，可能包含重复条目）。将 PID 写入一个 cgroup 的 **tasks** 文件，可将此进程移至该 cgroup。

cgroup.procs

包含在 cgroup 中运行的线程群组列表（由它们的 TGID 表示）。TGID 列表不一定是有序的，也不一定是特有的（也就是说，可能包含重复条目）。将 TGID 写入 cgroup 的 **cgroup.procs** 文件，可将此线程组群移至该 cgroup。

cgroup.event_control

与 cgroup 的通知 API 一起，允许 cgroup 的变更状态通知被发送。

notify_on_release

包含 Boolean 值，**1** 或者 **0**，分别可以启动和禁用释放代理的指令。如果 ***notify_on_release*** 启用，当 cgroup 不再包含任何任务时（即，cgroup 的 **tasks** 文件包含 PID，而 PID 被移除，致使文件变空），kernel 会执行 **release_agent** 文件的内容。通向此空 cgroup 的路径会作为释放代理的参数被提供。

notify_on_release 参数的默认值在 root cgroup 中是 **0**。所有非 root cgroup 从其父 cgroup 处继承 ***notify_on_release*** 的值。

release_agent （仅在 root group 中出现）

当 “notify on release” 被触发，它包含要执行的指令。一旦 cgroup 的所有进程被清空，并且 ***notify_on_release*** 标记被启用，kernel 会运行 **release_agent** 文件中的指令，并且提供通向被清空 cgroup 的相关路径（与 root cgroup 相关）作为参数。例如，释放代理可以用来自动移除空 cgroup，更多信息，请参阅例 A.4 “自动移除空 cgroup”。

例 A.4. 自动移除空 cgroup

参照以下步骤，可将空 cgroup 从 **cpu** cgroup 中自动移除：

1. 例如，创建一个 shell 脚本用来移除空 **cpu** cgroups，将其放入 **/usr/local/bin**，并使其可运行。

```
~]# cat /usr/local/bin/remove-empty-cpu-cgroup.sh
#!/bin/sh
rmdir /cgroup/cpu/$1
~]# chmod +x /usr/local/bin/remove-empty-cpu-cgroup.sh
```

   $1 变量包含到达已清空 cgroup 的相对路径。

2. 在 **cpu** cgroup，启动 ***notify_on_release*** 标签:

```
~]# echo 1 > /cgroup/cpu/notify_on_release
```

3. 在 **cpu** cgroup 中，指定一个可用的释放代理：

```
~]# echo "/usr/local/bin/remove-empty-cpu-cgroup.sh" >
/cgroup/cpu/release_agent
```

4. 测试您的配置，以确保已清空 cgroup 被正确移除：

```
cpu]# pwd; ls
/cgroup/cpu
cgroup.event_control  cgroup.procs  cpu.cfs_period_us
cpu.cfs_quota_us  cpu.rt_period_us  cpu.rt_runtime_us
cpu.shares  cpu.stat  libvirt  notify_on_release
release_agent  tasks
cpu]# cat notify_on_release
1
cpu]# cat release_agent
/usr/local/bin/remove-empty-cpu-cgroup.sh
cpu]# mkdir blue; ls
blue  cgroup.event_control  cgroup.procs  cpu.cfs_period_us
cpu.cfs_quota_us  cpu.rt_period_us  cpu.rt_runtime_us
cpu.shares  cpu.stat  libvirt  notify_on_release
release_agent  tasks
cpu]# cat blue/notify_on_release
1
cpu]# cgexec -g cpu:blue dd if=/dev/zero of=/dev/null
bs=1024k &
[1] 8623
cpu]# cat blue/tasks
8623
cpu]# kill -9 8623
cpu]# ls
cgroup.event_control  cgroup.procs  cpu.cfs_period_us
cpu.cfs_quota_us  cpu.rt_period_us  cpu.rt_runtime_us
cpu.shares  cpu.stat  libvirt  notify_on_release
release_agent  tasks
```

A.13. 附加资源

Kernel 子系统专项介绍

以下所有文件都位于 **/usr/share/doc/kernel-doc-<*kernel_version*>/Documentation/cgroups/** 目录中（由 *kernel-doc* 软件包提供）。

- **blkio** 子系统 —— **blkio-controller.txt**
- **cpuacct** 子系统 ——**cpuacct.txt**
- **cpuset** 子系统 —— **cpusets.txt**
- **devices** 子系统 —— **devices.txt**
- **freezer** 子系统 —— **freezer-subsystem.txt**
- **memory** 子系统 —— **memory.txt**
- **net_prio** 子系统 — **net_prio.txt**

另外，关于 **cpu** 子系统的更多信息，请参阅下列文件：

- 实时调度程序 — **/usr/share/doc/kernel-doc-<*kernel_version*>/Documentation/scheduler/sched-rt-group.txt**
- CFS 调度程序 — **/usr/share/doc/kernel-doc-<*kernel_version*>/Documentation/scheduler/sched-bwc.txt**

---

[1] 源代码由 Red Hat 的工程师 František Hrbata 提供。

[2] 源代码由 Red Hat 的工程师 František Hrbata 提供。

附录 B. 修订历史

修订 0.0-1.4.1	Mon Mar 16 2016	Chester Cheng

说明：7.1 版翻译、校对完成。
翻译、校对：李雪丹。
校对、编辑：傅同杰。
校对、责任编辑：鄭中。
附注：本简体中文版来自「**红帽公司・全球服务部**」与「**澳大利亚昆士兰大学・笔译暨口译研究生院**」之产学合作计划。若有疏漏之处，盼各方先进透过以下网址，给予支持指正：https://bugzilla.redhat.com/。

修订 0.0-1.4	Thu Feb 19 2015	Radek Bíba

7.1 GA 版本发行。Linux 容器移至独立手册中。

修订 0.0-1.0	Mon Jul 21 2014	Peter Ondrejka

修订 0.0-0.14	Mon May 13 2013	Peter Ondrejka

7.0 GA 版本发行