红帽企业版 Linux 6

资源管理指南

在 Red Hat Enterprise Linux 6 中管理系统

版 1

Last Updated: 2017-11-29

红帽企业版 Linux 6 资源管理指南

在 Red Hat Enterprise Linux 6 中管理系统
版 1

Martin Prpič
红帽 工程内容服务
mprpic@redhat.com

Rüdiger Landmann
红帽 工程内容服务
r.landmann@redhat.com

Douglas Silas
红帽 工程内容服务
dhensley@redhat.com

法律通告

摘要

在 Red Hat Enterprise Linux 6 中管理系统

目录

第 1 章 控制组群（CGROUP）简介

Red Hat Enterprise Linux 6 提供新的内核功能：*控制族群（control group）*，本手册中使用其简称 *cgroup*。Cgroup 可让您为系统中所运行任务（进程）的用户定义组群分配资源 -- 比如 CPU 时间、系统内存、网络带宽或者这些资源的组合。您可以监控您配置的 cgroup，拒绝 cgroup 访问某些资源，甚至在运行的系统中动态配置您的 cgroup。可将 **cgconfig**（"*控制组群配置*"）服务配置为在引导时启动，并重新建立您预先定义的 cgroup，这样可使其在重启过程中保留它们。

使用 cgroup，系统管理员可更具体地控制对系统资源的分配、优先顺序、拒绝、管理和监控。可更好地根据任务和用户分配硬件资源，提高总体效率。

1.1. 如何管理控制组群

Cgroup 是分层管理的，类似进程，且子 cgroup 会继承其上级 cgroup 的一些属性。但这两个模式也有不同。

Linux 进程模式

Linux 系统中的所有进程都是通用父进程 **init** 的子进程，该进程在引导时由内核执行并启动其它进程（这些进程会按顺序启动其子进程）。因为所有进程都归结到一个父进程，所以 Linux 进程模式是一个单一层级结构，或者树结构。

另外，**init** 之外的每个 Linux 进程都会继承其父进程的环境（比如 PATH 变量）[1]和某些属性（比如打开文件描述符）。

Cgroup 模式

Cgroup 与进程在以下方面类似：

- 它们是分级的，且
- 子 cgroup 会继承父 cgroup 的某些属性。

根本的不同是在某个系统中可同时存在不同的分级 cgroup。如果 Linux 进程模式是进程的单一树模式，那么 cgroup 模式是一个或者更多任务的独立、未连接树（例如：进程）。

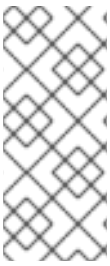
需要多个独立 cgroup 分级，因为每个分级都会附加到 *一个或者多个子系统*中。子系统[2]代表单一资源，比如 CPU 时间或者内存。Red Hat Enterprise Linux 6 提供 9 个 cgroup 子系统，根据名称和功能列出如下。

Red Hat Enterprise Linux 中的可用子系统

- **blkio** -- 这个子系统为块设备设定输入/输出限制，比如物理设备（磁盘，固态硬盘，USB 等等）。
- **cpu** -- 这个子系统使用调度程序提供对 CPU 的 cgroup 任务访问。
- **cpuacct** -- 这个子系统自动生成 cgroup 中任务所使用的 CPU 报告。
- **cpuset** -- 这个子系统为 cgroup 中的任务分配独立 CPU（在多核系统）和内存节点。
- **devices** -- 这个子系统可允许或者拒绝 cgroup 中的任务访问设备。
- **freezer** -- 这个子系统挂起或者恢复 cgroup 中的任务。
- **memory** -- 这个子系统设定 cgroup 中任务使用的内存限制，并自动生成由那些任务使用的内存资源报告。

- **net_cls** -- 这个子系统使用等级识别符（classid）标记网络数据包，可允许 Linux 流量控制程序（**tc**）识别从具体 cgroup 中生成的数据包。

- **ns** -- *名称空间*子系统。



注意

您可能在 cgroup 文献，比如 man page 或者内核文档中看到术语*资源控制器*或者*控制器*。这两个词与 “subsystem（子系统）”的含义相同，且基于这样的事实，即子系统通常调度资源或者在其所附属层级的 cgroup 中应用限制。

子系统（资源控制器）的定义非常普通：它是根据一组任务行动的东西，例如进程。

1.2. 子系统、层级、控制组群和任务的关系

请记住 cgroup 术语中系统进程称为任务。

这里有一些简单的规则管理子系统、cgroup 层级以及任务之间的关系，并给出那些规则的合理结果。

规则 1

任何单一子系统（比如 **cpu**）最多可附加到一个层级中。

*结果是，**cpu** 子系统永远无法附加到两个不同的层级。*

规则 2

单一层级可附加一个或者多个子系统。

*结果是，**cpu** 和 **memroy** 子系统（或者任意数目的子系统）都可附加到单一层级中，只要每个子系统不再附加到另一个层级即可。*

规则 3

每次在系统中创建新层级时，该系统中的所有任务都是那个层级的默认 cgroup（我们称之为 *root cgroup*）的初始成员。对于您创建的任何单一层级，该系统中的每个任务都可以是那个层级中*唯一一个* cgroup 的成员。单一任务可以是在多个 cgroup 中，只要每个 cgroup 都在不同的层级中即可。只要某个任务成为同一层级中第二个 cgroup 的成员，就会将其从那个层级的第一个 cgroup 中删除。一个任务永远不会同时位于同一层级的不同 cgroup 中。

*结果是，如果 **cpu** 和 **memory** 子系统都附加到名为 **cpu_and_mem** 的层级中，且 **net_cls** 子系统是附加到名为 **net** 的层级中，那么运行的 **httpd** 进程可以是 **cpu_and_mem** 中任意 cgroup 的成员，同时也是 **net** 中任意 cgroup 的成员。*

***httpd** 进程所在 **cpu_and_mem** 中的 cgroup 可将其 CPU 时间限制为分配给其它进程时间的一半，并将其内存用量限制为最多 **1024** MB。另外，**net** 中的 cgroup 还可将其传输速率限制为 **30** MB/秒。*

首次创建层级时，该系统中的每个任务都至少是一个 cgroup 的成员，即 root cgroup。因此每当使用 cgroup 时，每个系统任务总是至少在一个 cgroup 中。

规则 4

该系统中的任意进程（任务）都将自己分支创建子进程（任务）。该子任务自动成为其父进程所在 cgroup 的成员。然后可根据需要将该子任务移动到不同的 cgroup 中，但开始时它总是继承其父任务的 cgroup（进程术语中称其为“环境”）。

***cpu_and_mem** 层级中名为 **half_cpu_1gb_max** 的 cgroup 成员的任务，以及 **net** 层级中 cgroup*

`trans_rate_30` 的成员。当 `httpd` 进程将其自身分成几个分支时，其子进程会自动成为 `half_cpu_1gb_max` cgroup 和 `trans_rate_30` cgroup 的成员。它会完全继承其父任务所属的同一 cgroup。

此后，父任务和子任务就彼此完全独立：更改某个任务所属 cgroup 不会影响到另一个。同样更改父任务的 cgroup 也不会以任何方式影响其子任务。总之：所有子任务总是可继承其父任务的同一 cgroup 的成员关系，但之后可更改或者删除那些成员关系。

1.3. 资源管理实施

- 因为某个任务可属于任一层级中的单一 cgroup，所以只有一个方法可让单一子系统限制或者影响任务。这是合理的：是一个功能，而不是限制。
- 您可以将几个子系统分组在一起以便它们可影响单一层级中的所有任务。因为该层级中的 cgroup 有不同的参数设定，因此会对那些任务产生不同的影响。
- 有时可能需要*重构*层级。例如：从附加了几个子系统的层级中删除一个子系统，并将其附加到不同的层级中。
- 反正，如果从不同层级中分离子系统的需求降低，则您可以删除层级并将其子系统附加到现有层级中。
- 这个设计允许简单的 cgroup 使用，比如为单一层级中的具体任务设定几个参数，单一层级可以是只附加了 cpu 和 memeory 子系统的层级。
- 这个设计还允许高精度配置：系统中的每个任务（进程）都可以是每个层级的成员，每个层级都有单一附加的子系统。这样的配置可让系统管理员绝对控制每个单一任务的所有参数。

---

[1] 父进程可在将环境传递给子进程前更改它。

[2] 您应该了解在 libcgroup man page 和其它文档中，子系统也称*资源控制器*，或者*控制器*。

第 2 章 使用控制组群

使用 cgroup 的最简单的方法是安装 libcgroup 软件包，该软件包包含大量与 cgroup 有关的命令行工具及其相关 man page。您可以使用 shell 命令和可在任意系统中使用的工具挂载层级并设定 cgroup 参数（不保留）。但是，使用 libcgroup 提供的工具可简化过程并扩展功能。因此，本指南着重全面介绍 libcgroup 命令。在大多数情况下，我们会将对等的 shell 命令包括在内以便清楚阐述基础机理。我们建议您在可行的情况下尽量使用 libcgroup 命令。

注意

要使用 cgroup，首先请确定在您的系统中安装 libcgroup 软件包，方法为作为 root 运行：

```
~]# yum install libcgroup
```

2.1. CGCONFIG 服务

由 libcgroup 软件包安装的 **cgconfig** 服务可提供创建层级的方便方法，并在层级中附加子系统，且在那些层级中管理 cgroup。我们建议您使用 **cgconfig** 在您的系统中管理层级和 cgroup。

红帽企业版 Linux 6 不默认启动 **cgconfig** 服务。当使用 **chkconfig** 启动该服务时，它读取 cgroup 配置文件 -- **/etc/cgconfig.conf**。Cgroup 因此会在不同会话间重新创建并保留。根据配置文件的内容，**cgconfig** 可创建层级、挂载所需文件系统、创建 cgroup 以及为每个组群设定子系统参数。

libcgroup 软件包默认安装的 **/etc/cgconfig.conf** 文件为每个子系统创建并挂载独立层级，并在这些层级中附加子系统。

如果您停止 **cgconfig** 服务（使用 **service cgconfig stop** 命令），则会卸载它挂载的所有层级。

2.1.1. cgconfig.conf 文件

cgconfig.conf 文件包含两个主要类型的条目 -- *mount* 和 *group*。挂载条目生成并挂载层级并将其作为虚拟文件系统，同时将子系统附加到那些层级中。挂载条目使用以下语法定义：

```
mount {
    <controller> = <path>;
    …
}
```

示例用法请参考 例 2.1 “创建挂载条目”。

例 2.1. 创建挂载条目

以下示例为 **cpuset** 子系统创建层级：

```
mount {
    cpuset = /cgroup/cpu;
}
```

对等的 shell 命令：

```
~]# mkdir /cgroup/cpu
~]# mount -t cgroup -o cpu cpu /cgroup/cpu
```

组群条目创建 cgroup 并设定子系统参数。组群条目使用以下语法定义：

```
group <name> {
    [<permissions>]
    <controller> {
        <param name> = <param value>;
        …
    }
    …
}
```

请注意 **permissions** 部分是可选的。要为组群条目定义权限，请使用以下语法：

```
perm {
    task {
        uid = <task user>;
        gid = <task group>;
    }
    admin {
       uid = <admin name>;
       gid = <admin group>;
    }
}
```

示例用法请参考 例 2.2 “创建组群条目”：

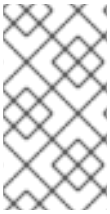
例 2.2. 创建组群条目

以下示例为 sql 守护进程创建 cgroup，可为 **sqladmin** 组群中的用户在 cgroup 中添加任务，并让 **root** 用户修改子系统参数：

```
group daemons/sql {
    perm {
        task {
            uid = root;
            gid = sqladmin;
        } admin {
            uid = root;
            gid = root;
        }
    } cpu {
        cpu.shares = 100;
    }
}
```

当与 例 2.1 “创建挂载条目” 中的挂载条目示例合并时，对等的 shell 命令为：

```
~]# mkdir -p /cgroup/cpu/daemons/sql
~]# chown root:root /cgroup/cpu/daemons/sql/*
~]# chown root:sqladmin /cgroup/cpu/daemons/sql/tasks
~]# echo 100 > /cgroup/cpu/daemons/sql/cpu.shares
```



注意

您必须重启 **cgconfig** 服务方可使 **/etc/cgconfig.conf** 中的更改生效：

```
~]# service cgconfig restart
```

当安装 cgroups 软件包时，会在 **/etc/cgconfig.conf** 中写入示例配置文件。每行开始的 # 符号将该行注释出来以便 **cgconfig** 服务忽略它。

2.2. 创建层级并附加子系统

警告

以下步骤覆盖从创建新层级到在其中附加子系统到内容，这些步骤假设还没有在您的系统中配置 cgroup。在这种情况下，这些步骤不会影响系统中的操作。更改有任务的 cgroup 中的可调参数可能会马上影响那些任务。本指南提示您它首次演示更改会影响一个或者多个任务的可调 cgroup 参数。

在已经配置了 cgroup 的系统中（可以是手动配置，或者使用 **cgconfig** 服务配置），这些命令会失败，除非您首先卸载可影响系统操作的现有层级。不要在产品系统中试验这些步骤。

要创建层级并在其中附加子系统，请作为 root 编辑 **/etc/cgconfig.conf** 文件的 **mount** 部分。**mount** 部分的条目有以下格式：

```
subsystem = /cgroup/hierarchy;
```

下一次启动 **cgconfig** 时，它会创建层级并为其附加子系统。

以下示例创建名为 **cpu_and_mem** 的层级，并附加 **cpu**、**cpuset**、**cpuacct** 和 **memory** 子系统。

```
mount {
    cpuset  = /cgroup/cpu_and_mem;
    cpu     = /cgroup/cpu_and_mem;
    cpuacct = /cgroup/cpu_and_mem;
    memory  = /cgroup/cpu_and_mem;
}
```

备用方法

您还可以使用 shell 命令和工具创建层级并在其中附加子系统。

作为 root 为该层级创建*挂载点*。在挂载点中包括 cgroup 名称：

```
~]# mkdir /cgroup/name
```

例如：

```
~]# mkdir /cgroup/cpu_and_mem
```

下一步，请使用 **mount** 命令挂载该层级并同时附加一个或者多个系统。例如：

```
~]# mount -t cgroup -o subsystems name /cgroup/name
```

其中 *subsystems* 是使用逗号分开的子系统列表，*name* 是层级名称。所有可用子系统的简要论述请参考 Red Hat Enterprise Linux 中的可用子系统，第 3 章 *子系统和可调参数* 中有详细的参考。

例 2.3. 使用 mount 命令附加子系统

在这个示例中，已经有名为 **/cgroup/cpu_and_mem** 的目录，它可以作为我们所创建层级的挂载点服务。我们将在名为 **cpu_and_mem** 的层级中附加 **cpu**、**cpuset** 和 **memory** 子系统，并在 **/cgroup/cpu_and_mem** 中 **mount cpu_and_mem** 层级：

```
~]# mount -t cgroup -o cpu,cpuset,memory cpu_and_mem /cgroup/cpu_and_mem
```

您可以使用 **lssubsys** [3]命令列出所有可用子系统及其当前挂载点（例如：挂载附加这些子系统的层级的位置）

```
~]# lssubsys -am
cpu,cpuset,memory /cgroup/cpu_and_mem
net_cls
ns
cpuacct
devices
freezer
blkio
```

这个输出表示：

- 在挂载到 **/cgroup/cpu_and_mem** 的层级中附加子系统 **cpu**、**cpuset** 和 **memory**，且
- 还没有在任何层级中附加子系统 **net_cls**、**ns**、**cpuacct**、**devices**、**freezer** 和 **blkio**，因缺少相应的挂载点。

2.3. 在现有层级中附加或者删除子系统

要在现有层级中添加子系统，从现有层级中取消层级或者将其移动到不同的层级中，请作为 root 编辑 **/etc/cgconfig.conf** 文件的 **mount** 部分，使用 第 2.2 节 “创建层级并附加子系统” 中所述的语法。当 **cgconfig** 下次启动时，它会根据您指定的层级识别那些子系统。

备用方法

要在现有层级中取消附加子系统，请重新挂载该层级。请在 **mount** 命令中包括额外的子系统以及 **remount** 选项。

例 2.4. 重新挂载层级添加子系统

lssubsys 命令显示在 **cpu_and_mem** 层级中附加 **cpu**、**cpuset** 和 **memory** 子系统：

```
~]# lssubsys -am
```

```
cpu,cpuset,memory /cgroup/cpu_and_mem
net_cls
ns
cpuacct
devices
freezer
blkio
```

我们使用 **remount** 选项重新挂载 **cpu_and_mem** 层级，并在子系统列表中包含 **cpuacct**：

```
~]# mount -t cgroup -o remount,cpu,cpuset,cpuacct,memory cpu_and_mem
/cgroup/cpu_and_mem
```

lssubsys 命令现在显示附加到 **cpu_and_mem** 层级中的 **cpuacct**：

```
~]# lssubsys -am
cpu,cpuacct,cpuset,memory /cgroup/cpu_and_mem
net_cls
ns
devices
freezer
blkio
```

同样，您可以重新挂载该层级并使用 -o 选项忽略子系统名称从现有层级中分离子系统。例如：要分离 **cpuacct** 子系统，只要您重新挂载并忽略它即可：

```
~]# mount -t cgroup -o remount,cpu,cpuset,memory cpu_and_mem
/cgroup/cpu_and_mem
```

2.4. 卸载层级

您还可以使用 **umount** 命令*卸载* cgroup 中的层级：

```
~]# umount /cgroup/name
```

例如：

```
~]# umount /cgroup/cpu_and_mem
```

如果该层级目前为空（即它只包含 root cgroup），则在卸载它时会取消激活该层级。如果该层级包含任意其他 cgroup，该层级在内核中仍保持活跃，即使不再挂载它也是如此。

要删除层级，请确定您在卸载该层级前删除所有子 cgroup，或者使用 **cgclear** 命令，它可在层级非空时也可取消激活层级 -- 请参考 第 2.11 节 “卸载控制组群”。

2.5. 创建控制组群

请使用 **cgcreate** 命令创建 cgroup。**cgcreate** 的语法为：**cgcreate -t *uid*:*gid* -a *uid*:*gid* -g *subsystems*:*path***，其中：

- **-t**（可选）- 指定用户（使用用户 ID，uid）和组群（使用组群 ID，gid）以便让这个 cgroup 拥有 **tasks** 伪文件。这个用户可在该 cgroup 中添加任务。

  注意

  请注意：从 cgroup 中删除任务的唯一方法是将其移动到不同的 cgroup中。要移动任务，该用户必须有*目的* cgroup 的写访问。对源 cgroup 的写访问并不重要。

- **-a**（可选）- 指定用户（使用用户 ID，uid）和组群（使用组群 ID，gid）以便这个 cgroup 拥有 **tasks** 外的所有伪文件。这个用户可修改这个 cgroup 中的任务对系统资源的访问。

- **-g** -- 指定在其中创建 cgroup 的层级，格式为与那些层级关联的用逗号分开的 *subsystems* 列表。如果这个列表中的子系统在不同的层级中，则要在每个层级中都创建该组群。层级列表后是一个冒号，然后是与该层级有关的子组群 *path*。不要在该 path 中包含层级挂载点。

  例如：目录 **/cgroup/cpu_and_mem/lab1/** 中的 cgroup 称为 **lab1** -- 其路径已唯一确定，因为对于给定的子系统最多有一个层级。还请注意该组群可由创建该 cgroup 的现有层级中的所有子系统控制，即使没有在 **cgcreate** 命令中指定这些子系统 -- 请参考 例 2.5 “cgcreate 用法”。

因为同一层级中的所有 cgroup 有相同的控制器，该子组群与其父 cgroup 有相同的控制器。

> **例 2.5. cgcreate 用法**
>
> 请考虑在 **cpu_and_mem** 层级中一同挂载 **cpu** 和 **memory** 子系统的系统，并将 **net_cls** 控制器挂载到名为 **net** 的另一个层级中。我们现在运行：
>
> ```
> ~]# cgcreate -g cpu,net_cls:/test-subgroup
> ```
>
> **cgcreate** 命令创建两个组群，名为 **test-subgroup**，一个位于 **cpu_and_mem** 层级，一个位于 **net** 层级。**cpu_and_mem** 层级中的 **test-subgroup** 组群由 **memory** 子系统控制，即使在 **cgcreate** 命令中没有指定它也是如此。

备用方法

请使用 **mkdir** 命令直接创建 cgroup 的子组群：

```
~]# mkdir /cgroup/hierarchy/name/child_name
```

例如：

```
~]# mkdir /cgroup/cpuset/lab1/group1
```

2.6. 删除控制组群

使用 **cgdelete** 删除 cgroup，其语法与 **cgcreate** 类似。运行 **cgdelete *subsystems*:*path***，其中：

- *subsystems* 是用逗号分开的子系统列表。
- *path* 是到与该层级相对 root 的 cgroup 的路径。

例如：

```
~]# cgdelete cpu,net_cls:/test-subgroup
```

cgdelete 还可使用 **-r** 选项递归删除所有子组群。

删除 cgroup 时，将其所有任务都移动到它的父组群中。

2.7. 设置参数

在可修改相关的 cgroup 的用户帐户中运行 **cgset** 命令设定子系统参数。例如：如果有 **/cgroup/cpuset/group1**，则请使用以下命令指定这个组群可访问的 CPU：

```
cpuset]# cgset -r cpuset.cpus=0-1 group1
```

cgset 的语法为：**cgset -r *parameter=value path_to_cgroup*** ，其中：

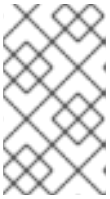
- *parameter* 是要设定的参数，该参数与给定 cgroup 的目录中的文件对应。
- *value* 是为参数设定的值
- *path_to_cgroup* 是到相对*该层级 root 的* cgroup 路径。例如：如果设定 root 组群的参数（如有 **/cgroup/cpuacct/** 文件），请运行：

  ```
  cpuacct]# cgset -r cpuacct.usage=0 /
  ```

  另外，因为 **.** 与 root 组群相关（即 root 组群本身），您还可运行：

  ```
  cpuacct]# cgset -r cpuacct.usage=0 .
  ```

  备注：**/** 是首选语法。

  注意

  只能为该 root 组群设定少量参数（比如上面的示例中演示的 **cpuacct.usage** 参数）。这是因为 root 组群拥有所有现有资源，因此定义某些参数限制现有进程就没有任何意义，例如 **cpuset.cpu** 参数。

  要设定 root 组群的子组群 **group1** 参数，请运行：

  ```
  cpuacct]# cgset -r cpuacct.usage=0 group1
  ```

  该组群名称结尾的斜杠（例如：**cpuacct.usage=0 group1/**）是可选的。

您可以使用 **cgset** 设定的值可能取决于在具体层级中设定的较大的值。例如：如果将 **group1** 限制为只能使用系统中的 CPU 0，则您无法将 **group1/subgroup1** 设定为使用 CPUs 0 和 1，或者只使用 CPU 1。

您还可以使用 **cgset** 将一个 cgroup 中的参数复制到另一个现有 cgroup 中，例如：

```
~]# cgset --copy-from group1/ group2/
```

使用 **cgset** 复制参数的语法为：**cgset --copy-from *path_to_source_cgroup* *path_to_target_cgroup***，其中：

- *path_to_source_cgroup* 是相对该层级中 root 组群，到要复制其参数的 cgroup 的路径。
- *path_to_target_cgroup* 是相对该层级 root 组群的目的 cgroup 的路径。

请确定您在从一个组群将参数复制到另一个组群前为不同子系统设定强制参数，否则命令会失败。有关强制参数的详情请参考 重要 - 强制参数。

备用方法
要直接在 cgroup 中设置参数，请使用 **echo** 命令将值插入相关子系统伪文件。例如：这个命令可将值 **0-1** 插入 cgroup **group1** 的 **cpuset.cpus** 伪文件中：

```
~]# echo 0-1 > /cgroup/cpuset/group1/cpuset.cpus
```

在此设定这个值，则这个 cgroup 中的任务就限制在该系统的 CPU 0 和 1 中。

2.8. 将某个进程移动到控制组群中

您还可以运行 **cgclassify** 命令将进程移动到 cgroup 中：

```
~]# cgclassify -g cpu,memory:group1 1701
```

cgclassify 的语法为：**cgclassify -g *subsystems*:*path_to_cgroup* *pidlist***，其中：

- *subsystems* 是用逗号分开的子系统列表，或者 ***** 启动与所有可用子系统关联的层级中的进程。请注意：如果在多个层级中有同名的 cgroup，则 **-g** 选项会将该进程移动到每个组群中。请确定在拥有您在此指定子系统的每个层级中都有该 cgroup。
- *path_to_cgroup* 是到其层级中的 cgroup 的路径
- *pidlist* 是用空格分开的*进程识别符*（PID）列表

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** 守护进程正在运行，则会根据 **/etc/cgrules.conf** 中的设置将子进程分配到 cgroup 中。该进程本身则仍保留在启动它的 cgroup 中。

使用 **cgclassify**，您可以同时移动多个进程。例如：这个命令将 PID 为 **1701** 和 **1138** 的进程移动到 cgroup 中的 **group1/**：

```
~]# cgclassify -g cpu,memory:group1 1701 1138
```

请注意要移动的 PID 间要用空格分开，且应该在不同的层级中指定这些组群。

备用方法
要将进程直接移动到 cgroup 中，请将其 PID 写入该 cgroup 中的 **tasks** 文件中。例如：要将 PID 为 **1701** 的进程移动到位于 **/cgroup/lab1/group1/** 的 cgroup 中：

```
~]# echo 1701 > /cgroup/lab1/group1/tasks
```

2.8.1. cgred 守护进程

Cgred 是一个守护进程，它可根据在 **/etc/cgrules.conf** 文件中设定的参数将任务移动到 cgroup 中。**/etc/cgrules.conf** 文件中的条目可以使用以下两个格式之一：

- *user hierarchies control_group*
- *user:command hierarchies control_group*

例如：

```
maria    devices   /usergroup/staff
```

这个条目指定任何属于名为 **maria** 用户的进程根据在 **/usergroup/staff** cgroup 中指定的参数访问设备子系统。要将具体命令与具体 cgroup 关联，请添加 *command* 参数，如下：

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

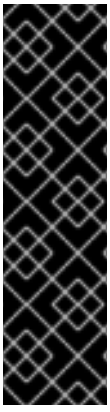
该条目现在指定何时名为 **maria** 的用户使用 **ftp** 命令，自动将该进程移动到包含 **devices** 子系统的层级中的 **/usergroup/staff/ftp** cgroup 中。请注意：该守护进程只有在符合适当的条件后才可将该进程移动到该 cgroup 中。因此，**ftp** 可能会在错误的组群中短暂运行。再有，如果该进程在错误组群中迅速生出子进程，则不会移动这些子进程。

/etc/cgrules.conf 文件中的条目可包括以下额外符号：

- **@** - 当在 *user* 使用前缀时，代表是一个组群而不是单独用户。例如：**@admins** 是 **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。
- **%** - 代表与以上行中项目相同的项目。例如：

```
@adminstaff   devices   /admingroup
@labstaff   %   %
```

2.9. 在控制组群中启动一个进程



重要

有些子系统拥有强制参数，您在可以将任务移动到使用那些子系统的 cgroup 前必须设定这些参数。例如：在您将任务移动到使用 **cpuset** 子系统的 cgroup 前，必须为那个 cgroup 定义 ***cpuset.cpus*** 和 ***cpuset.mems*** 参数。

本章的示例演示了命令的正确语法，但只适用于在此示例中为所有控制器设定了相关强制参数的系统。如果您还没有配置相关控制器，您就无法直接将本章中的命令示例直接用于您的系统。

给定子系统的强制参数的描述请参考 第 3.10 节 “附加资源”。

您还可以运行 **cgexec** 命令在 cgroup 中启动进程。例如：这个命令启动了 **group1** cgroup 中的 **lynx** 网页浏览器，目的是限制 **cpu** 子系统为那个组群造成的负担。

```
~]# cgexec -g cpu:group1 lynx http://www.redhat.com
```

cgexec 语法为：**cgexec -g *subsystems*:*path_to_cgroup* *command* *arguments***，其中：

- *subsystems* 是用逗号分开的子系统列表或者 ***** 启动与所有可用子系统关联的层级中的进程。请注意：如 第 2.7 节 “设置参数” 所述的 **cgset**，如果在多个层级中有同名的 cgroup，**-g** 选项会在每个组群中创建进程。请确定您在拥有在此指定的子系统的层级中都有该 cgroup。
- *path_to_cgroup* 是到与该层级相关的 cgroup 的路径。
- *command* 是要运行的命令
- *arguments* 是该命令所有参数

您还可以在 *command* 前面添加 **-- sticky** 将所有子进程放在同一 cgroup 中。如果您没有设定这个选项，且 **cgred** 守护进程正在运行，则将根据在 **/etc/cgrules.conf** 中的设置将子进程分配到 cgroup 中。而该进程本身仍保留在启动它的 cgroup 中。

备用方法
当您启动新进程时，它会继承其父进程的组群。因此在具体 cgroup 中启动进程的备选方法是将您的 shell 进程移动到那个组群中（请参考 第 2.8 节 “将某个进程移动到控制组群中”），然后在那个 shell 中启动该进程。例如：

```
~]# echo $$ > /cgroup/lab1/group1/tasks
lynx
```

请注意：退出 **lynx** 后，您现有 shell 中仍在 **group1** cgroup 中。因此更好的方法应为：

```
~]# sh -c "echo \$$ > /cgroup/lab1/group1/tasks && lynx"
```

2.9.1. 在控制组群中启动服务

您可在某个 cgroup 中启动某些服务。在 cgroup 中启动的服务必须：

- 使用 **/etc/sysconfig/*servicename*** 文件
- 使用 **/etc/init.d/functions** 的 **daemon()** 功能启动该服务

要在某个 cgroup 中启动合格服务，请在 **/etc/sysconfig** 中编辑其文件，使该文件包含格式如下的条目：**CGROUP_DAEMON="*subsystem*:*control_group*"**，其中 *subsystem* 是与具体层级关联的子进程，*control_group* 是那个层级中的 cgroup。例如：

```
CGROUP_DAEMON="cpuset:daemons/sql"
```

2.10. 获得有关控制组群的信息

2.10.1. 查找某个进程

要查找某个进程所属 cgroup，请运行：

```
~]$ ps -O cgroup
```

或者如果您知道该进程的 PID，请运行：

```
~]$ cat /proc/PID/cgroup
```

2.10.2. 查找子系统

要查找可在您内核中使用的子系统以及如何将其挂载到层级中，请运行：

```
~]$ cat /proc/cgroups
```

或者查找具体子系统的挂载点，请运行：

```
~]$ lssubsys -m subsystems
```

其中 *subsystems* 是您感兴趣的子系统列表。请注意：**lssubsys -m** 命令只返回每个层级的顶级挂载点。

2.10.3. 查找层级

我们建议您在 **/cgroup** 挂载层级。假设在您的系统中是这种情况，列出或者浏览包含层级组群的目录中的内容。如果在您的系统中安装了 **tree**，运行该程序获得所有层级概述以及其中的 cgroup：

```
~]$ tree /cgroup/
```

2.10.4. 查找控制组群

要查找某个系统的 cgroup，请运行：

```
~]$ lscgroup
```

您可以指定控制程序和路径限制具体层级的输出，格式为 ***controller*:*path***。例如：

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

只列出附加 **cpuset** 子系统的层级中的 **adminusers** cgroup 的子组群。

2.10.5. 显示控制组群的参数

要显示具体 cgroup 的参数，请运行：

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

其中 *parameter* 是包含子系统值的伪文件，*list_of_cgroups* 是用空格分开的 cgroup 列表。例如：

```
~]$ cgget -r cpuset.cpus -r memory.limit_in_bytes lab1 lab2
```

显示 cgroup **lab1** 和 **lab2** 的 **cpuset.cpus** 和 **memory.limit_in_bytes** 值。

如果您不知道参数名称，请使用类似命令：

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

2.11. 卸载控制组群

警告

cgclear 目录将破坏所有层级中的所有 cgroup。如果您没有在配置文件中保存这些层级，则您无法再次构建它们。

要清除整个 cgroup 文件系统，请使用 **cgclear** 命令。

将该 cgroup 中的所有任务重新分配到该层级的 root 节点中；删除所有 cgroup；从该系统中卸载这个文件系统；这样就破坏了所有之前挂载的层级。最后，实际上是删除了挂载该 cgroup 文件系统的目录。

注意

使用 **mount** 命令创建 cgroup（与使用 **cgconfig** 服务创建它们相反），结果是在 **/etc/mtab** 文件（挂载的文件系统表）中生成一个条目。这个更改还在 **/proc/mounts** 有所体现。但是使用 **cgclear** 命令卸载 cgroup，与 **cgconfig** 命令一同使用直接内核界面则不会在 **/etc/mtab** 文件中有所体现，而只是在 **/proc/mounts** 文件中写入新信息。因此使用 **cgclear** 命令卸载 cgroup 仍可在 **/etc/mtab** 文件中看出来，且在随后执行 **mount** 命令时显示。所有挂载 cgroup 的准确列表，建议您参考 **/proc/mounts** 文件。

2.12. 附加资源

cgroup 命令最权威的文档是 libcgroup 软件包提供的手册页。这部分的数字在以下 man page 列表中指定。

libcgroup Man Page

- **man 1 cgclassify** -- **cgclassify** 命令是用来将运行的任务移动到一个或者多个 cgroup。

  man 1 cgclear -- **cgclear** 命令是用来删除层级中的所有 cgroup。

  man 5 cgconfig.conf -- 在 **cgconfig.conf** 文件中定义 cgroup。

  man 8 cgconfigparser -- **cgconfigparser** 命令解析 **cgconfig.conf** 文件和并挂载层级。

  man 1 cgcreate -- **cgcreate** 在层级中创建新 cgroup。

  man 1 cgdelete -- **cgdelete** 命令删除指定的 cgroup。

  man 1 cgexec -- **cgexec** 命令在指定的 cgroup 中运行任务。

  man 1 cgget -- **cgget** 命令显示 cgroup 参数。

  man 5 cgred.conf -- **cgred.conf** 是 **cgred** 服务的配置文件。

  man 5 cgrules.conf -- **cgrules.conf** 包含用来决定何时任务术语某些 cgroup 的规则。

man 8 cgrulesengd -- **cgrulesengd** 在 cgroup 中发布任务。

man 1 cgset -- **cgset** 命令为 cgroup 设定参数。

man 1 lscgroup -- **lscgroup** 命令列出层级中的 cgroup。

man 1 lssubsys -- **lssubsys** 命令列出包含指定子系统的层级。

---

[3] **lssubsys** 是 libcgroup 命令提供的工具之一。您必须安装 libcgroup 方可使用这个工具：如果您无法运行 **lssubsys**，请参考 第 2 章 *使用控制组群*。

第 3 章 子系统和可调参数

*子系统*是识别 cgroup 的内核模块。通常它们是为不同 cgroup 分配各种系统登记的资源控制器。但是可为其它与内核的互动编辑子系统，这些互动需要以不同方式对待不同的进程组群。开发新子系统的*应用程序编程界面*（API）文档位于内核文件的 **cgroups.txt** 中，该文件安装在您系统的 **/usr/share/doc/kernel-doc-*kernel-version*/Documentation/cgroups/**（由 kernel-doc 软件包提供）。cgroup 文档的最新版本还可在 http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt 中找到。请注意：最新文档中的功能可能不与您系统中安装的内核匹配。

用于 cgroup 的包含子系统参数的*状态对象*在 cgroup 的虚拟文件系统中是以*伪文件*出现。这些伪文件可由 shell 命令或者与其对等的系统调用操作。例如：**cpuset.cpus** 是用来指定可允许 cgroup 访问哪些 CPU。如果 **/cgroup/cpuset/webserver** 是用于系统中运行的网页服务器的 cgroup，则我们会运行以下命令：

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

将值 **0,2** 写入 **cpuset.cpus** 伪文件，并因此将 PID 为列在 **/cgroup/cpuset/webserver/tasks/** 中的任务限制为只使用系统中的 CPU 0 和 CPU 2。

3.1. BLKIO

块 I/O（**blkio**）子系统控制并监控 cgroup 中的任务对块设备的 I/O 访问。在部分这样的伪文件中写入值可限制访问或者带宽，且从这些伪文件中读取值可提供 I/O 操作信息。

blkio.weight

指定 cgroup 默认可用访问块 I/O 的相对比例（*加权*），范围在 **100** 到 **1000**。这个值可由具体设备的 **blkio.weight_device** 参数覆盖。例如：如果将 cgroup 访问块设备的默认加权设定为 **500**，请运行：

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

blkio.weight_device

指定对 cgroup 中可用的具体设备 I/O 访问的相对比例（*加权*），范围是 **100** 到 **1000**。这个值可由为设备指定的 **blkio.weight** 参数覆盖。这个值的格式为*major:minor weight*，其中 *major* 和 *minor* 是在*《Linux 分配的设备》*中指定的设备类型和节点数，我们也称之为 *Linux 设备列表*，您可以参考 http://www.kernel.org/doc/Documentation/devices.txt。例如：如果为访问 **/dev/sda** 的 cgroup 分配加权 **500**，请运行：

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

在*《Linux 分配的设备》*注释中，**8:0** 代表 **/dev/sda**。

blkio.time

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* 是在 *Linux 分配的设备*中指定的设备类型和节点数，*time* 是时间长度，单位为毫秒（ms）。

blkio.sectors

报告使用 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*、*minor* 是在 *Linux 分配的设备*中指定的设备类型和节点数，*sectors* 是磁盘扇区数。

blkio.io_service_bytes

报告使用 cgroup 转换到具体设备或者由具体设备中转换出的字节数。条目有四个字段：*major*、*minor*、*operation* 和 *bytes*。*Major* 和 *minor* 是在 *Linux 分配的设备*中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* 是转换的字节数。

blkio.io_serviced

报告使用 cgroup 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* 是在 *Linux 分配的设备*中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.io_service_time

报告使用 cgroup 在具体设备中的 I/O 操作请求发送和请求完成之间的时间。条目有四个字段：*major*、*minor*、*operation* 和 *time*。*Major* 和 *minor* 是在 *Linux 分配的设备*中指定的设备类型和节点数，*operation* 代表操作类型（**read**、**write**、**sync** 或者 **async**），*time* 是时间长度，单位为纳秒（ns）。使用纳秒为单位报告而不是较大的单位是要使报告即使对固态设备也是有意义的。

blkio.io_wait_time

报告在具体设备中 cgroup 为调度程序队列中等待的 I/O 操作时间总计。当您解读这个报告时，请注意：

- 报告的时间可以比消耗的时间更长，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。要查找该组群作为整体而消耗的等待时间，请使用 **blkio.group_wait_time**。
- 如果该设备有 **queue_depth** > 1，则报告的时间只包括向该设备发送请求之前的时间，而不包括在该设备重新提出请求时等待服务的任何时间。

条目有四个字段：*major*、*minor*、*operation* 和 *bytes*。*Major* 和 *minor* 是在*《Linux 分配的设备》*中指定的设备类型和节点数，*operation* 代表操作类型（**read**、**write**、**sync** 或者 **async**），*time* 是时间长度，单位为纳秒（ns）。使用纳秒报告而不是更大的单位可让这个报告对固态设备也有意义。

blkio.io_merged

报告使用 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* 代表操作类型（**read**、**write**、**sync** 或者 **async**）。

blkio.io_queued

报告 cgroup 为 I/O 操作排队的请求次数。条目有两个字段：*number* 和 *operation*。*Number* 是请求次数，*operation* 代表操作类型（**read**、**write**、**sync** 或者 **async**）。

blkio.avg_queue_size

报告在该组群存在的整个过程中，cgroup 的 I/O 操作的平均队列大小。每次这个 cgroup 获得一个时间单位时都对该队列大小进行采样。请注意这个报告只有在将系统设定为 **CONFIG_DEBUG_BLK_CGROUP=y** 时可用。

blkio.group_wait_time

报告 cgroup 等待每个队列的时间总计（单位为纳秒 -- ns）。每次这个 cgroup 的队列获得一个时间单位时就会更新这个报告，因此如果您在 cgroup 等待时间单位时读取这个伪文件，则该报告将不会包含等待当前队列中的操作的时间。请注意这个报告只有在将系统设定为 **CONFIG_DEBUG_BLK_CGROUP=y** 时可用。

blkio.empty_time

报告 cgroup 在没有任何等待处理请求时花费的时间总计（单位为纳秒 -- ns）。每次这个 cgroup 有等待处理请求时都会更新这个报告，因此如果您在 cgroup 没有任何等待处理请求是读取这个伪文件，则该报告中不会包含消耗在当前空状态中的时间。请注意这个报告只有在将该系统设定为 **CONFIG_DEBUG_BLK_CGROUP=y** 时可用。

blkio.idle_time

报告调度程序在 cgroup 等待比已经在其它队列或者来自其它组群请求更好的请求时显示闲置的时间总计（单位为纳秒 -- ns）。每次该组群不显示闲置时就会更新这个报告，因此如果您在 cgroup 闲置时读取这个伪文件，则该报告将不会包括消耗在当前闲置状态的时间。请注意，只有在将系统设定为 **CONFIG_DEBUG_BLK_CGROUP=y** 时这个报告才可用。

blkio.dequeue

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* 是在 *Linux 分配的设备*中指定的设备类型和节点数，*number* 是将该组群从队列中移除的次数。请注意这个报告只有在将系统设定为 **CONFIG_DEBUG_BLK_CGROUP=y** 时可用。

blkio.reset_stats

重新设定在其它伪文件中记录的统计数据。在这个文件中写入一个整数为这个 cgroup 重新设定统计数据。

3.2. CPU

cpu 子系统调度对 cgroup 的 CPU 访问。可根据以下参数调度对 CPU 资源的访问，每个参数都独立存在于 cgroup 虚拟文件系统的*伪文件*中：

cpu.shares

包含用来指定在 cgroup 中的任务可用的相对共享 CPU 时间的整数值。例如：在两个 cgroup 中都将 **cpu.shares** 设定为 **1** 的任务将有相同的 CPU 时间，但在 cgroup 中将 **cpu.shares** 设定为 **2** 的任务可使用的 CPU 时间是在 cgroup 中将 **cpu.shares** 设定为 **1** 的任务可使用的 CPU 时间的两倍。

cpu.rt_runtime_us

以微秒（µs，这里以“us”代表）为单位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。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 中的任务应该可以每 5 秒中可有 4 秒时间访问 CPU 资源，请将 **cpu.rt_runtime_us** 设定为 **4000000**，并将 **cpu.rt_period_us** 设定为 **5000000**。

cpu.rt_period_us

以微秒（µs，这里以“us”代表）为单位指定在某个时间段中 cgroup 对 CPU 资源访问重新分配的频率。如果某个 cgroup 中的任务应该每 5 秒钟有 4 秒时间可访问 CPU 资源，则请将 **cpu.rt_runtime_us** 设定为 **4000000**，并将 **cpu.rt_period_us** 设定为 **5000000**。

3.3. CPUACCT

CPU Accounting（**cpuacct**）子系统自动生成 cgroup 中任务所使用的 CPU 资源报告，其中包括子组群中的任务。三个可用报告为：

cpuacct.stat

报告这个 cgroup 及其子组群中的任务使用用户模式和系统（内核）模式消耗的 CPU 循环数（单位由系统中的 **USER_HZ** 定义）。

cpuacct.usage

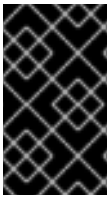
报告这个 cgroup 中所有任务（包括层级中的低端任务）消耗的总 CPU 时间（纳秒）。

cpuacct.usage_percpu

报告这个 cgroup 中所有任务（包括层级中的低端任务）在每个 CPU 中消耗的 CPU 时间（以纳秒为单位）。

3.4. CPUSET

cpuset 子系统为 cgroup 分配独立 CPU 和内存节点。可根据用以下参数指定每个 cpuset，每个参数都在控制组群虚拟文件系统中有单独的*伪文件*：



重要

有些子系统拥有强制参数，您在可以将任务移动到使用那些子系统的 cgroup 前必须设定这些参数。例如：在您将任务移动到使用 **cpuset** 子系统的 cgroup 前，必须为那个 cgroup 定义 ***cpuset.cpus*** 和 ***cpuset.mems*** 参数。

cpuset.cpus（强制）

指定允许这个 cgroup 中任务访问的 CPU。这是一个用逗号分开的列表，格式为 ASCII，使用小横线（"-"）代表范围。例如：

```
0-2,16
```

代表 CPU 0、1、2 和 16。

cpuset.mems（强制）

指定允许这个 cgroup 中任务可访问的内存节点。这是一个用逗号分开的列表，格式为 ASCII，使用小横线（"-"）代表范围。例如：

```
0-2,16
```

代表内存节点 0、1、2 和 16。

cpuset.memory_migrate

包含用来指定当 **cpuset.mems** 中的值更改时是否应该将内存中的页迁移到新节点的标签（**0** 或者 **1**）。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，即使在 **cpuset.mems** 中现已不再指定这个节点。如果启用（**1**），则该系统会将页迁移到由 **cpuset.mems** 指定的新参数中的内存节点中，可能的情况下保留其相对位置 - 例如：原来由 **cpuset.mems** 指定的列表中第二个节点中的页将会重新分配给现在由 **cpuset.mems** 指定的列表的第二个节点中，如果这个位置是可用的。

cpuset.cpu_exclusive

包含指定是否其它 cpuset 及其上、下级族群可共享为这个 cpuset 指定的 CPU 的标签（**0** 或者 **1**）。默认情况下（**0**）CPU 不是专门分配给某个 cpuset 的。

cpuset.mem_exclusive

包含指定是否其它 cpuset 可共享为这个 cpuset 指定的内存节点的标签（**0** 或者 **1**）。默认情况下（**0**）内存节点不是专门分配给某个 cpuset 的。专门为某个 cpuset 保留内存节点（**1**）与使用 **cpuset.mem_hardwall** 启用内存 hardwall 功能是一致的。

cpuset.mem_hardwall

包含指定是否应将内存页面的内核分配限制在为这个 cpuset 指定的内存节点的标签（**0** 或者 **1**）。默认情况下为 **0**，属于多个用户的进程共享页面和缓冲。启用 hardwall 时（**1**）每个任务的用户分配应保持独立。

cpuset.memory_pressure

包含运行在这个 cpuset 中产生的平均*内存压力*的只读文件。启用 **cpuset.memory_pressure_enabled** 时，这个伪文件中的值会自动更新，否则伪文件包含的值为 **0**。

cpuset.memory_pressure_enabled

包含指定系统是否应该计算这个 cgroup 中进程所生成*内存压力*的标签（**0** 或者 **1**）。计算出的值会输出到 **cpuset.memory_pressure**，且代表进程试图释放使用中内存的比例，报告为尝试每秒再生内存的整数值再乘 1000。

cpuset.memory_spread_page

包含指定是否应将文件系统缓冲平均分配给这个 cpuset 的内存节点的标签（**0** 或者 **1**）。默认情况为 **0**，不尝试为这些缓冲平均分配内存页面，且将缓冲放置在运行生成缓冲的进程的同一节点中。

cpuset.memory_spread_slab

包含指定是否应在 cpuset 间平均分配用于文件输入/输出操作的内核缓存板的标签（**0** 或者 **1**）。默认情况是 **0**，即不尝试平均分配内核缓存板，并将缓存板放在生成这些缓存的进程所运行的同一节点中。

cpuset.sched_load_balance

包含指定是否在这个 cpuset 中跨 CPU 平衡负载内核的标签（**0** 或者 **1**）。默认情况是 **1**，即内核将超载 CPU 中的进程移动到负载较低的 CPU 中以便平衡负载。

请注意：如果在任意上级 cgroup 中启用负载平衡，则在 cgroup 中设定这个标签没有任何效果，因为已经在较高一级 cgroup 中处理了负载平衡。因此，要在 cgroup 中禁用负载平衡，还要在该层级的每一个上级 cgroup 中禁用负载平衡。这里您还应该考虑是否应在所有平级 cgroup 中启用负载平衡。

cpuset.sched_relax_domain_level

包含 **-1** 到小正数间的整数，它代表内核应尝试平衡负载的 CPU 宽度范围。如果禁用了 **cpuset.sched_load_balance**，则该值毫无意义。

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	为负载平衡使用系统默认值

值	效果
0	不执行直接负载平衡；负载平衡只是阶段性的
1	在同一核中的跨线程直接负载平衡
2	在同一软件包中的跨线程直接负载平衡
3	在同一节点或者刀片中的跨线程直接负载平衡
4	在不使用统一内存访问（NUMA）构架中跨多个 CPU 的直接负载平衡
5	在使用统一内存访问（NUMA）构架中跨多个 CPU 的直接负载平衡

3.5. DEVICES

devices 子系统允许或者拒绝 cgroup 中的任务访问设备。

重要

在红帽企业版 Linux 6 中将设备白名单列表（**devices**）子系统视为技术预览。

目前红帽企业版 Linux 6 订阅服务还不支持*技术预览*功能，可能功能并不完备，且通常不适合产品使用。但红帽在操作系统中包含这些功能是为了方便用户，并提供更广泛的功能。您会发现这些功能可能在非产品环境中很有用，同时还请提供反馈一件和功能建议，以便今后全面支持这个技术预览。

devices.allow

指定 cgroup 中的任务可访问的设备。每个条目有四个字段：*type*、*major*、*minor* 和 *access*。*type*、*major* 和 *minor* 字段中使用的值对应 *Linux 分配的设备*，也称 *Linux 设备列表*中指定的设备类型和节点数，这两本书可见于 http://www.kernel.org/doc/Documentation/devices.txt。

type

type 可以是以下三个值之一：

- **a** - 应用所有设备，可以是*字符设备*，也可以是*块设备*
- **b** - 指定块设备
- **c** - 指定字符设备

major, minor

major 和 *minor* 是由《*Linux 分配设备*》指定的设备节点数。主设备号和副设备号使用冒号分开。例如：**8** 是主设备号，指定 SCSI 磁盘驱动器；副设备号 **1** 指定第一个 SCSI 磁盘驱动器中的第一个分区；因此**8:1** 完整指定这个分区，对应位于 **/dev/sda1** 的一个文件系统。

***** 可代表所有主要和次要设备节点，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序列：

- **r** - 允许任务从指定设备中读取
- **w** - 允许任务写入指定设备
- **m** - 允许任务生成还不存在的设备文件

例如：当将 *access* 指定为 **r** 时，则只能从指定设备中读取任务，但当将 *access* 指定为 **rw** 时，则既可从该设备中读取任务，也可向该设备中写入任务。

devices.deny

指定 cgroup 中任务不能访问的设备。条目语法与 **devices.allow** 一致。

devices.list

报告为这个 cgroup 中的任务设定访问控制的设备。

3.6. FREEZER

freezer 子系统挂起或者恢复 cgroup 中的任务。

freezer.state

freezer.state 有三个可能的值：

- **FROZEN** -- 挂起该 cgroup 中的任务。
- **FREEZING** -- 该系统正在挂起该 cgroup 中的任务。
- **THAWED** -- 已经恢复该 cgroup 中的任务。

要挂起具体进程：

1. 将那个进程移动到附加了 **freezer** 子系统的层级的 cgroup 中。
2. 停滞那个具体 cgroup 以便挂起其中包含的这个进程。

不可能将进程移动到挂起（frozen）的 cgroup 中。

请注意可将 **FROZEN** 和 **THAWED** 值写入 **freezer.state**，但无法写入 **FREEZING**，只能读取它。

3.7. MEMORY

memory 子系统自动生成 cgroup 中任务使用的内存资源报告，并设定由那些任务使用的内存限制：

memory.stat

报告大范围内存统计，如下表所述：

表 3.1. memory.stat 报告的值

统计	描述
cache	页缓存，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 映射的文件大小，包括 **tmpfs**（**shmem**），单位为字节
pgpgin	存入内存中的页数
pgpgout	从内存中读出的页数
swap	swap 用量，单位为字节
active_anon	在活跃的最近最少使用（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	不活跃的 LRU 列表中的匿名和 swap 缓存，包括 **tmpfs**（**shmem**），单位为字节
active_file	活跃 LRU 列表中的 file-backed 内存，以字节为单位
inactive_file	不活跃 LRU 列表中的 file-backed 内存，以字节为单位
unevictable	无法再生的内存，以字节为单位
hierarchical_memory_limit	包含 **memory** cgroup 的层级的内存限制，单位为字节
hierarchical_memsw_limit	包含 **memory** cgroup 的层级的内存加 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，还可在其下级 cgroup 中报告。例如：**swap** 报告 cgroup 的 swap 用量，**total_swap** 报告该 cgroup 及其所有子组群的 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** 的文件缓存 + swap 缓存

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

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

memory.usage_in_bytes

报告该 cgroup中进程使用的当前总内存用量（以字节为单位）。

memory.memsw.usage_in_bytes

报告该 cgroup 中进程使用的当前内存用量和 swap 空间总和（以字节为单位）。

memory.max_usage_in_bytes

报告该 cgroup 中进程使用的最大内存用量（以字节为单位）。

memory.memsw.max_usage_in_bytes

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

memory.limit_in_bytes

设定用户内存的最大量（包括文件缓存）。如果没有指定单位，则将该数值理解为字节。但是可以使用前缀代表更大的单位 - **k** 或者 **K** 代表千字节，**m** 或者 **M** 代表 MB，**g** 或者 **G** 代表 GB。

您不能使用 **memory.limit_in_bytes** 限制 root cgroup；您只能在该层级中较低的组群中应用这些值。

在 **memory.limit_in_bytes** 中写入 **-1** 删除现有限制。

memory.memsw.limit_in_bytes

设定最大内存与 swap 用量之和。如果没有指定单位，则将该值解读为字节。但是可以使用前缀代表更大的单位 - **k** 或者 **K** 代表千字节，**m** 或者 **M** 代表 MB，**g** 或者 **G** 代表 GB。

您不能使用 **memory.memsw.limit_in_bytes** 限制 root cgroup；您只能在该层级中较低的组群中应用这些值。

在 **memory.memsw.limit_in_bytes** 中写入 **-1** 删除现有限制。

memory.failcnt

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

memory.memsw.failcnt

报告内存加 swap 空间限制达到在 **memory.memsw.limit_in_bytes** 设定的值的次数。

memory.force_empty

当设定为 **0** 时，会清空这个 cgroup 中任务所使用的所有页面的内存。这个接口只可在 cgroup 中没有任务时使用。如果无法清空内存，则在可能的情况下将其移动到上级 cgroup 中。删除 cgroup 前请使用 **memory.force_empty** 以避免将不再使用的页面缓存移动到它的上级 cgroup 中。

memory.swappiness

将内核倾向设定为换出这个 cgroup 中任务所使用的进程内存，而不是从页缓冲中再生页面。这也是在 **/proc/sys/vm/swappiness** 中设定的使用同一方法为整个系统设定的内核倾向。默认值为 **60**。低于这个值会降低内核换出进程内存的倾向，将其设定为 **0** 则完全不会为 cgroup 中的任务换出进程内存。高于这个值将提高内核换出进程内存的倾向，大于 **100** 时内核将开始换出作为这个 cgroup 中进程的地址空间一部分的页面。

请注意那个值为 **0** 不会阻止换出进程内存；系统缺少内存时仍可能发生换出内存，这是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()** 而不时 cgroup。

您不能更改以下组群的 swappiness：

- root cgroup，它使用在 **/proc/sys/vm/swappiness** 中设定的 swappiness。
- 有属于它的子组群的 cgroup。

memory.use_hierarchy

包含指定是否应将内存用量计入 cgroup 层级的吞吐量的标签（**0** 或者 **1**）。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（**0**）是子系统不从任务的子进程中再生内存。

3.8. NET_CLS

net_cls 子系统使用等级识别符（classid）标记网络数据包，可允许 Linux 流量控制程序（**tc**）识别从具体 cgroup 中生成的数据包。可将流量控制程序配置为给不同 cgroup 中的数据包分配不同的优先权。

net_cls.classid

net_cls.classid 包含一个说明流量控制 *句柄* 的十六进制的值。例如：**0x1001** 代表通常写成 **10:1** 的句柄，这是 iproute2 使用的格式。

这些句柄的格式为：**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**。

参考 **tc** 的 man page 了解如何配置流量控制程序使用 **net_cls** 添加到网络数据包中的句柄。

3.9. NS

ns 子系统提供了一个将进程分组到不同 *名称空间* 的方法。在具体名称空间中，进程可彼此互动，但会与在其它名称空间中运行的进程隔绝。这些分开的名称空间在用于操作系统级别的虚拟化时，有时也称之为 *容器*。

3.10. 附加资源

具体子系统内核文档

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**

附录 A. 修订记录

修订 1-3.400 **2013-10-31** **Rüdiger Landmann**
Rebuild with publican 4.0.0

修订 1-3 **2012-07-18** **Anthony Towns**
Rebuild for Publican 3.0

修订 1.0-5 **Thu May 19 2011** **Martin Prpič**
《资源管理指南 的 Red Hat Enterprise Linux 6.1 正式发行本。

修订 1.0-4 **Tue Mar 1 2011** **Martin Prpič**
修复的多个示例 -- BZ#667623、BZ#667676、BZ#667699
cgclear 命令说明 -- BZ#577101
lssubsystem 命令说明 -- BZ#678517
冻结进程 -- BZ#677548

修订 1.0-3 **Wed Nov 17 2010** **Rüdiger Landmann**
修正重新挂载示例 -- BZ#612805

修订 1.0-2 **Thu Nov 11 2010** **Rüdiger Landmann**
删除预先发布的反馈说明

修订 1.0-1 **Wed Nov 10 2010** **Rüdiger Landmann**
修正来自 QE 的反馈 -- BZ#581702 和 BZ#612805

修订 1.0-0 **Tue Nov 9 2010** **Rüdiger Landmann**
正式发行本的完整功能版本