

---

# **Linux Translations Documentation**

**The kernel development community**

**Jan 15, 2023**



## **CONTENTS**

<b>1</b>	<b>中文翻译</b>	<b>1</b>
<b>2</b>	<b>繁體中文翻譯</b>	<b>633</b>
<b>3</b>	<b>Traduzione italiana</b>	<b>843</b>
<b>4</b>	<b>한국어 번역</b>	<b>1083</b>
<b>5</b>	<b>日本語訳</b>	<b>1151</b>
<b>6</b>	<b>Disclaimer</b>	<b>1165</b>
	<b>Bibliography</b>	<b>1167</b>
	<b>Index</b>	<b>1169</b>



## 中文翻译

---

**Note:** 翻译计划: 内核中文文档欢迎任何翻译投稿, 特别是关于内核用户和管理员指南部分。

---

这是中文内核文档树的顶级目录。内核文档, 就像内核本身一样, 在很大程度上是一项正在进行的工作; 当我们努力将许多分散的文件整合成一个连贯的整体时尤其如此。另外, 随时欢迎您对内核文档进行改进; 如果您想提供帮助, 请加入 vger.kernel.org 上的 linux-doc 邮件列表。

顺便说下, 中文文档也需要遵守内核编码风格, 风格中中文和英文的主要不同就是中文的字符标点占用两个英文字符宽度, 所以, 当英文要求不要超过每行 100 个字符时, 中文就不要超过 50 个字符。另外, 也要注意 - ‘,’ =’ 等符号与相关标题的对齐。在将补丁提交到社区之前, 一定要进行必要的 checkpatch.pl 检查和编译测试。

## \* 许可证文档

下面的文档介绍了 Linux 内核源代码的许可证 (GPLv2)、如何在源代码树中正确标记单个文件的许可证、以及指向完整许可证文本的链接。

- [Linux 内核许可规则](#)

## \* 用户文档

下面的手册是为内核用户编写的——即那些试图让它在给定系统上以最佳方式工作的用户。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解, 而不是作为一个分支。因此, 如果您对此文件有任何意见或更新, 请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题, 请联系该文件的译者, 或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** [.../..../admin-guide/index](#)

**Translator** Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

## \* Linux 内核用户和管理员指南

下面是一组随时间添加到内核中的面向用户的文档的集合。到目前为止，还没有一个整体的顺序或组织 - 这些材料不是一个单一的，连贯的文件！幸运的话，情况会随着时间的推移而迅速改善。

这个初始部分包含总体信息，包括描述内核的 README，关于内核参数的文档等。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/admin-guide/README.rst

译者 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

## Linux 内核 5.x 版本 <<http://kernel.org/>>

以下是 Linux 版本 5 的发行注记。仔细阅读它们，它们会告诉你这些都是什么，解释如何安装内核，以及遇到问题时该如何做。

## 什么是 Linux？

Linux 是 Unix 操作系统的克隆版本，由 Linus Torvalds 在一个松散的网络黑客（Hacker，无贬义）团队的帮助下从头开始编写。它旨在实现兼容 POSIX 和单一 UNIX 规范。

它具有在现代成熟的 Unix 中应当具有的所有功能，包括真正的多任务处理、虚拟内存、共享库、按需加载、共享的写时拷贝（COW）可执行文件、恰当的内存管理以及包括 IPv4 和 IPv6 在内的复合网络栈。

Linux 在 GNU 通用公共许可证，版本 2 (GNU GPLv2) 下分发，详见随附的 COPYING 文件。

## 它能在什么样的硬件上运行？

虽然 Linux 最初是为 32 位的 x86 PC 机（386 或更高版本）开发的，但今天它也能运行在（至少）Compaq Alpha AXP、Sun SPARC 与 UltraSPARC、Motorola 68000、PowerPC、PowerPC64、ARM、Hitachi SuperH、Cell、IBM S/390、MIPS、HP PA-RISC、Intel IA-64、DEC VAX、AMD x86-64 Xtensa 和 ARC 架构上。

Linux 很容易移植到大多数通用的 32 位或 64 位体系架构，只要它们有一个分页内存管理单元（PMMU）和一个移植的 GNU C 编译器（gcc；GNU Compiler Collection，GCC 的一部分）。Linux 也被移植到许多没有 PMMU 的体系架构中，尽管功能显然受到了一定的限制。Linux 也被移植到了其自己上。现在可以将内核作为用户空间应用程序运行——这被称为用户模式 Linux（UML）。

## 文档

因特网上和书籍上都有大量的电子文档，既有 Linux 专属文档，也有与一般 UNIX 问题相关的文档。我建议在任何 Linux FTP 站点上查找 LDP（Linux 文档项目）书籍的文档子目录。本自述文件并不是关于系统的文档：有更好的可用资源。

- 因特网上和书籍上都有大量的（电子）文档，既有 Linux 专属文档，也有与普通 UNIX 问题相关的文档。我建议在任何有 LDP（Linux 文档项目）书籍的 Linux FTP 站点上查找文档子目录。本自述文件并不是关于系统的文档：有更好的可用资源。
- 文档/子目录中有各种自述文件：例如，这些文件通常包含一些特定驱动程序的内核安装说明。请阅读 Documentation/process/changes.rst 文件，它包含了升级内核可能会导致的问题的相关信息。

## 安装内核源代码

- 如果您要安装完整的源代码，请把内核 tar 档案包放在您有权限的目录中（例如您的主目录）并将其解包：

```
xz -cd linux-5.x.tar.xz | tar xvf -
```

将“X”替换成最新内核的版本号。

**【不要】**使用 /usr/src/linux 目录！这里有一组库头文件使用的内核头文件（通常是不完整的）。它们应该与库匹配，而不是被内核的变化搞得一团糟。

- 您还可以通过打补丁在 5.x 版本之间升级。补丁以 xz 格式分发。要通过打补丁进行安装，请获取所有较新的补丁文件，进入内核源代码（linux-5.x）的目录并执行：

```
xz -cd ../patch-5.x.xz | patch -p1
```

请**【按顺序】**替换所有大于当前源代码树版本的“x”，这样就可以了。您可能想要删除备份文件（文件名类似 xxx~ 或 xxx.orig），并确保没有失败的补丁（文件名类似 xxx# 或 xxx.rej）。如果有，不是你就是我犯了错误。

与 5.x 内核的补丁不同，5.x.y 内核（也称为稳定版内核）的补丁不是增量的，而是直接应用于基本的 5.x 内核。例如，如果您的基本内核是 5.0，并且希望应用 5.0.3 补丁，则不应先应用 5.0.1 和 5.0.2 的补丁。类似地，如果您运行的是 5.0.2 内核，并且希望跳转到 5.0.3，那么在应用 5.0.3 补丁之前，必须首先撤销 5.0.2 补丁（即 `patch -R`）。更多关于这方面的内容，请阅读 [Documentation/process/applying-patches.rst](#)。

或者，脚本 `patch-kernel` 可以用来自动化这个过程。它能确定当前内核版本并应用找到的所有补丁：

```
linux/scripts/patch-kernel linux
```

上面命令中的第一个参数是内核源代码的位置。补丁是在当前目录应用的，但是可以将另一个目录指定为第二个参数。

- 确保没有过时的.o 文件和依赖项：

```
cd linux  
make mrproper
```

现在您应该已经正确安装了源代码。

## 软件要求

编译和运行 5.x 内核需要各种软件包的最新版本。请参考 [Documentation/process/changes.rst](#) 来了解最低版本要求以及如何升级软件包。请注意，使用过旧版本的这些包可能会导致很难追踪的间接错误，因此不要以为在生成或操作过程中出现明显问题时可以只更新包。

## 为内核建立目录

编译内核时，默认情况下所有输出文件都将与内核源代码放在一起。使用 `make O=output/dir` 选项可以为输出文件（包括`.config`）指定备用位置。例如：

```
kernel source code: /usr/src/linux-5.x  
build directory: /home/name/build/kernel
```

要配置和构建内核，请使用：

```
cd /usr/src/linux-5.x  
make O=/home/name/build/kernel menuconfig  
make O=/home/name/build/kernel  
sudo make O=/home/name/build/kernel modules_install install
```

请注意：如果使用了 `O=output/dir` 选项，那么它必须用于 `make` 的所有调用。

## 配置内核

即使只升级一个小版本，也不要跳过此步骤。每个版本中都会添加新的配置选项，如果配置文件没有按预定设置，就会出现奇怪的问题。如果您想以最少的工作量将现有配置升级到新版本，请使用 `make oldconfig`，它只会询问您新配置选项的答案。

- 其他配置命令包括：

<code>"make config"</code>	纯文本界面。
<code>"make menuconfig"</code>	基于文本的彩色菜单、选项列表和对话框。
<code>"make nconfig"</code>	增强的基于文本的彩色菜单。
<code>"make xconfig"</code>	基于 Qt 的配置工具。
<code>"make gconfig"</code>	基于 GTK+ 的配置工具。
<code>"make oldconfig"</code>	基于现有的 <code>./.config</code> 文件选择所有选项，并询问新配置选项。
<code>"make olddefconfig"</code>	类似上一个，但不询问直接将新选项设置为默认值。
<code>"make defconfig"</code>	根据体系架构，使用 <code>arch/\$arch/defconfig</code> 或 <code>arch/\$arch/configs/\${PLATFORM}_defconfig</code> 中的默认选项值创建 <code>./.config</code> 文件。
<code>"make \${PLATFORM}_defconfig"</code>	使用 <code>arch/\$arch/configs/\${PLATFORM}_defconfig</code> 中的默认选项值创建一个 <code>./.config</code> 文件。 用“ <code>make help</code> ”来获取您体系架构中所有可用平台的列表。
<code>"make allyesconfig"</code>	通过尽可能将选项值设置为“y”，创建一个 <code>./.config</code> 文件。
<code>"make allmodconfig"</code>	通过尽可能将选项值设置为“m”，创建一个 <code>./.config</code> 文件。
<code>"make allnoconfig"</code>	通过尽可能将选项值设置为“n”，创建一个 <code>./.config</code> 文件。
<code>"make randconfig"</code>	通过随机设置选项值来创建 <code>./.config</code> 文件。
<code>"make localmodconfig"</code>	基于当前配置和加载的模块 ( <code>lsmod</code> ) 创建配置。禁用已加载的模块不需要的任何模块选项。
要为另一台计算机创建 <code>localmodconfig</code> ，请将该计算机	

的 `lsmod` 存储到一个文件中，并将其作为 `lsmod` 参数传入。

此外，通过在参数 `LMC_KEEP` 中指定模块的路径，可以将模块保留在某些文件夹或 `kconfig` 文件中。

```
target$ lsmod > /tmp/my_lsmod
target$ scp /tmp/my_lsmod host:/tmp

host$ make LSMOD=/tmp/my_lsmod \
      LMC_KEEP="drivers/usb:drivers/gpu:fs" \
      localmodconfig
```

上述方法在交叉编译时也适用。

"`make localyesconfig`" 与 `localmodconfig` 类似，只是它会将所有模块选项转换为内置 (`=y`)。你可以同时通过 `LMC_KEEP` 保留模块。

"`make kvm_guest.config`" 为 `kvm` 客户机内核支持启用其他选项。

"`make xen.config`" 为 `xen dom0` 客户机内核支持启用其他选项。

"`make tinyconfig`" 配置尽可能小的内核。

更多关于使用 Linux 内核配置工具的信息，见文档 `Documentation/kbuild/kconfig.rst`。

- `make config` 注意事项：

- 包含不必要的驱动程序会使内核变大，并且在某些情况下会导致问题：探测不存在的控制器卡可能会混淆其他控制器。
- 如果存在协处理器，则编译了数学仿真的内核仍将使用协处理器：在这种情况下，数学仿真永远不会被使用。内核会稍微大一点，但不管是否有数学协处理器，都可以在不同的机器上工作。
- “kernel hacking” 配置细节通常会导致更大或更慢的内核（或两者兼而有之），甚至可以通过配置一些例程来主动尝试破坏坏代码以发现内核问题，从而降低内核的稳定性 (`kmalloc()`)。因此，您可能应该用于研究“开发”、“实验”或“调试”特性相关问题。

## 编译内核

- 确保您至少有 `gcc 5.1` 可用。有关更多信息，请参阅 `Documentation/process/changes.rst`。

请注意，您仍然可以使用此内核运行 `a.out` 用户程序。

- 执行 `make` 来创建压缩内核映像。如果您安装了 `lilo` 以适配内核 `makefile`，那么也可以进行 `make install`，但是您可能需要先检查特定的 `lilo` 设置。

实际安装必须以 `root` 身份执行，但任何正常构建都不需要。无须徒然使用 `root` 身份。

- 如果您将内核的任何部分配置为模块，那么还必须执行 `make modules_install`。
- 详细的内核编译/生成输出：

通常，内核构建系统在相当安静的模式下运行（但不是完全安静）。但是有时您或其他内核开发人员需要看到编译、链接或其他命令的执行过程。为此，可使用“verbose（详细）”构建模式。向 `make` 命令传递 `V=1` 来实现，例如：

```
make V=1 all
```

如需构建系统也给出内个目标重建的愿意，请使用 `V=2`。默认为 `V=0`。

- 准备一个备份内核以防出错。对于开发版本尤其如此，因为每个新版本都包含尚未调试的新代码。也要确保保留与该内核对应的模块的备份。如果要安装与工作内核版本号相同的新内核，请在进行 `make modules_install` 安装之前备份 `modules` 目录。

或者，在编译之前，使用内核配置选项“`LOCALVERSION`”向常规内核版本附加一个唯一的后缀。`LOCALVERSION` 可以在“General Setup”菜单中设置。

- 为了引导新内核，您需要将内核映像（例如编译后的`…/linux/arch/x86/boot/bzImage`）复制到常规可引导内核的位置。
- 不再支持在没有 LILO 等启动装载程序帮助的情况下直接从软盘引导内核。

如果从硬盘引导 Linux，很可能使用 LILO，它使用`/etc/lilo.conf` 文件中指定的内核映像文件。内核映像文件通常是`/vmlinuz`、`/boot/vmlinuz`、`/bzImage` 或 `/boot/bzImage`。使用新内核前，请保存旧映像的副本，并复制新映像覆盖旧映像。然后您【必须重新运行 LILO】来更新加载映射！否则，将无法启动新的内核映像。

重新安装 LILO 通常需要运行`/sbin/LILO`。您可能希望编辑`/etc/lilo.conf` 文件为旧内核映像指定一个条目（例如`/vmlinuz.old`）防止新的不能正常工作。有关更多信息，请参阅 LILO 文档。

重新安装 LILO 之后，您应该就已经准备好了。关闭系统，重新启动，尽情享受吧！

如果需要更改内核映像中的默认根设备、视频模式等，请在适当的地方使用启动装载程序的引导选项。无需重新编译内核即可更改这些参数。

- 使用新内核重新启动并享受它吧。

## 若遇到问题

- 如果您发现了一些可能由于内核缺陷所导致的问题，请检查 `MAINTAINERS`（维护者）文件看看是否有人与令您遇到麻烦的内核部分相关。如果无人在此列出，那么第二个最好的方案就是把它们发给我（[torvalds@linux-foundation.org](mailto:torvalds@linux-foundation.org)），也可能发送到任何其他相关的邮件列表或新闻组。
- 在所有的缺陷报告中，【请】告诉我们您在说什么内核，如何复现问题，以及您的设置是什么的（使用您的常识）。如果问题是新的，请告诉我；如果问题是旧的，请尝试告诉我您什么时候首次注意到它。
- 如果缺陷导致如下消息：

```
unable to handle kernel paging request at address C0000010
Oops: 0002
EIP: 0010:XXXXXXXX
eax: XXXXXXXX ebx: XXXXXXXX ecx: XXXXXXXX edx: XXXXXXXX
esi: XXXXXXXX edi: XXXXXXXX ebp: XXXXXXXX
ds: xxxx es: xxxx fs: xxxx gs: xxxx
Pid: xx, process nr: xx
XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX
```

或者类似的内核调试信息显示在屏幕上或在系统日志里，请【如实】复制它。可能对你来说转储（dump）看起来不可理解，但它确实包含可能有助于调试问题的信息。转储上方的文本也很重要：它说明了内核转储代码的原因（在上面的示例中，是由于内核指针错误）。更多关于如何理解转储的信息，请参见 Documentation/admin-guide/bug-hunting.rst。

- 如果使用 CONFIG\_KALLSYMS 编译内核，则可以按原样发送转储，否则必须使用 `ksymoops` 程序来理解转储（但通常首选使用 CONFIG\_KALLSYMS 编译）。此实用程序可从 <https://www.kernel.org/pub/linux/utils/kernel/ksymoops/> 下载。或者，您可以手动执行转储查找：
- 在调试像上面这样的转储时，如果您可以查找 EIP 值的含义，这将非常有帮助。十六进制值本身对我或其他任何人都没有太大帮助：它会取决于特定的内核设置。您应该做的是从 EIP 行获取十六进制值（忽略 0010:），然后在内核名字列表中查找它，以查看哪个内核函数包含有问题的地址。

要找到内核函数名，您需要找到与显示症状的内核相关联的系统二进制文件。就是文件“linux/vmlinux”。要提取名字列表并将其与内核崩溃中的 EIP 进行匹配，请执行：

```
nm vmlinux | sort | less
```

这将为您提供一个按升序排序的内核地址列表，从中很容易找到包含有问题的地址的函数。请注意，内核调试消息提供的地址不一定与函数地址完全匹配（事实上，这是不可能的），因此您不能只“grep”列表：不过列表将为您提供每个内核函数的起点，因此通过查找起始地址低于你正在搜索的地址，但后一个函数的高于的函数，你会找到您想要的。实际上，在您的问题报告中加入一些“上下文”可能是一个好主意，给出相关的上下几行。

如果您由于某些原因无法完成上述操作（如您使用预编译的内核映像或类似的映像），请尽可能多地告诉我您的相关设置信息，这会有所帮助。有关详细信息请阅读 ‘Documentation/admin-guide/reporting-issues.rst’。

- 或者，您可以在正在运行的内核上使用 `gdb`（只读的；即不能更改值或设置断点）。为此，请首先使用`-g` 编译内核；适当地编辑 `arch/x86/Makefile`，然后执行 `make clean`。您还需要启用 `CONFIG_PROC_FS`（通过 `make config`）。

使用新内核重新启动后，执行 `gdb vmlinux /proc/kcore`。现在可以使用所有普通的 `gdb` 命令。查找系统崩溃点的命令是 `l *0XXXXXXXX`（将 `xxx` 替换为 EIP 值）。

用 `gdb` 无法调试一个当前未运行的内核是由于 `gdb`（错误地）忽略了编译内核的起始偏移量。

Todolist:

- kernel-parameters

- devices
- sysctl/index

本节介绍 CPU 漏洞及其缓解措施。

Todolist:

- hw-vuln/index

下面的一组文档，针对的是试图跟踪问题和 bug 的用户。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/admin-guide/reporting-issues.rst

译者 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

## 报告问题

### 简明指南（亦即太长不看）

您面临的是否为同系列稳定版或长期支持内核的普通内核的回归？是否仍然受支持？请搜索 [LKML 内核邮件列表](#) 和 [Linux 稳定版邮件列表](#) 存档中匹配的报告并加入讨论。如果找不到匹配的报告，请安装该系列的最新版本。如果它仍然出现问题，报告给稳定版邮件列表 ([stable@vger.kernel.org](mailto:stable@vger.kernel.org))。

在所有其他情况下，请尽可能猜测是哪个内核部分导致了问题。查看 [MAINTAINERS](#) 文件，了解开发人员希望如何得知问题，大多数情况下，报告问题都是通过电子邮件和抄送相关邮件列表进行的。检查报告目的地的存档中是否已有匹配的报告；也请搜索 [LKML](#) 和网络。如果找不到可加入的讨论，请安装 [最新的主线内核](#)。如果仍存在问题，请发送报告。

问题已经解决了，但是您希望看到它在一个仍然支持的稳定版或长期支持系列中得到解决？请安装其最新版本。如果它出现了问题，那么在主线中搜索修复它的更改，并检查是否正在回传（backporting）或者已放弃；如果两者都没有，那么可询问处理更改的人员。

**通用提醒：**当安装和测试上述内核时，请确保它是普通的（即：没有补丁，也没有使用附加模块）。还要确保它是在一个正常的环境中构建和运行，并且在问题发生之前没有被污染（tainted）。

在编写报告时，要涵盖与问题相关的所有信息，如使用的内核和发行版。在碰见回归时，尝试给出引入它的更改的提交 ID，二分可以找到它。如果您同时面临 Linux 内核的多个问题，请分别报告每个问题。

一旦报告发出，请回答任何出现的问题，并尽可能地提供帮助。这包括通过不时重新测试新版本并发送状态更新来推动进展。

### 如何向内核维护人员报告问题的逐步指南

上面的简明指南概述了如何向 Linux 内核开发人员报告问题。对于已经熟悉向自由和开源软件（FLOSS）项目报告问题的人来说，这可能是他们所需要的全部内容。对于其他人，本部分更为详细，并一步一步地描述。为了便于阅读，它仍然尽量简洁，并省略了许多细节；这些在逐步指南后的参考章节中进行了描述，该章节更详细地解释了每个步骤。

注意：本节涉及的方面比简明指南多，顺序也稍有不同。这符合你的利益，以确保您尽早意识到看起来像 Linux 内核毛病的问题可能实际上是由其他原因引起的。这些步骤可以确保你最终不会觉得在这一过程中投入的时间是浪费：

- 您是否面临硬件或软件供应商提供的 Linux 内核的问题？那么基本上您最好停止阅读本文档，转而向您的供应商报告问题，除非您愿意自己安装最新的 Linux 版本。寻找和解决问题往往需要后者。
- 使用您喜爱的网络搜索引擎对现有报告进行粗略搜索；此外，请检查 [Linux 内核邮件列表（LKML）](#) 的存档。如果找到匹配的报告，请加入讨论而不是发送新报告。
- 看看你正在处理的问题是否为回归问题、安全问题或非常严重的问题：这些都是需要在接下来的一些步骤中特别处理的“高优先级问题”。
- 确保不是内核环境导致了您面临的问题。
- 创建一个新的备份，并将系统修复和恢复工具放在手边。
- 确保您的系统不会通过动态构建额外的内核模块来增强其内核，像 DKMS 这样的解决方案可能在您不知情的情况下就在本地进行了这样的工作。
- 当问题发生时，检查您的内核是否被“污染”，因为使内核设置这个标志的事件可能会导致您面临的问题。
- 粗略地写下如何重现这个问题。如果您同时处理多个问题，请为每个问题单独写注释，并确保它们在新启动的系统上独立出现。这是必要的，因为每个问题都需要分别报告给内核开发人员，除非它们严重纠缠在一起。
- 如果您正面临稳定版或长期支持版本线的回归（例如从 5.10.4 更新到 5.10.5 时出现故障），请查看后文“报告稳定版和长期支持内核线的回归”小节。
- 定位可能引起问题的驱动程序或内核子系统。找出其开发人员期望的报告的方式和位置。注意：大多数情况下不会是 [bugzilla.kernel.org](https://bugzilla.kernel.org)，因为问题通常需要通过邮件发送给维护人员和公共邮件列表。
- 在缺陷追踪器或问题相关邮件列表的存档中彻底搜索可能与您的问题匹配的报告。如果你发现了一些相关讨论，请加入讨论而不是发送新的报告。

在完成这些准备之后，你将进入主要部分：

- 除非您已经在运行最新的“主线”Linux 内核，否则最好在报告流程前安装它。在某些情况下，使用最新的“稳定版”Linux 进行测试和报告也是可以接受的替代方案；在合并窗口期间，这实际上可能是最

好的方法，但在开发阶段最好还是暂停几天。无论你选择什么版本，最好使用“普通”构建。忽略这些建议会大大增加您的报告被拒绝或忽略的风险。

- 确保您刚刚安装的内核在运行时不会“污染”自己。
- 在您刚刚安装的内核中复现这个问题。如果它没有出现，请查看下方只发生在稳定版和长期支持内核的问题的说明。
- 优化你的笔记：试着找到并写出最直接的复现问题的方法。确保最终结果包含所有重要的细节，同时让第一次听说的人容易阅读和理解。如果您在此过程中学到了一些东西，请考虑再次搜索关于该问题的现有报告。
- 如果失败涉及“panic”、“Oops”、“warning”或“BUG”，请考虑解码内核日志以查找触发错误的代码行。
- 如果您的问题是回归问题，请尽可能缩小引入问题时的范围。
- 通过详细描述问题来开始编写报告。记得包括以下条目：您为复现而安装的最新内核版本、使用的 Linux 发行版以及关于如何复现该问题的说明。如果可能，将内核构建配置 (.config) 和 dmesg 的输出放在网上的某个地方，并链接到它。包含或上传所有其他可能相关的信息，如 Oops 的输出/截图或来自 lspci 的输出。一旦你写完了这个主要部分，请在上方插入一个正常长度的段落快速概述问题和影响。再在此之上添加一个简单描述问题的句子，以得到人们的阅读。现在给出一个更短的描述性标题或主题。然后就可以像 MAINTAINERS 文件告诉你的那样发送或提交报告了，除非你在处理一个“高优先级问题”：它们需要按照下面“高优先级问题的特殊处理”所述特别关照。
- 等待别人的反应，继续推进事情，直到你能够接受这样或那样的结果。因此，请公开和及时地回应任何询问。测试提出的修复。积极地测试：至少重新测试每个新主线版本的首个候选版本 (RC)，并报告你的结果。如果出现拖延，就友好地提醒一下。如果你没有得到任何帮助或者未能满意，请试着自己帮助自己。

## 报告稳定版和长期支持内核线的回归

如果您发现了稳定版或长期支持内核版本线中的回归问题并按上述流程跳到这里，那么请阅读本小节。即例如您在从 5.10.4 更新到 5.10.5 时出现了问题（从 5.9.15 到 5.10.5 则不是）。开发人员希望尽快修复此类回归，因此有一个简化流程来报告它们：

- 检查内核开发人员是否仍然维护你关心的 Linux 内核版本线：去 [kernel.org](http://kernel.org) 的首页，确保此特定版本线的最新版没有 “[EOL]” 标记。
- 检查 [Linux 稳定版邮件列表](#) 中的现有报告。
- 从特定的版本线安装最新版本作为纯净内核。确保这个内核没有被污染，并且仍然存在问题，因为问题可能已经在那被修复了。如果您第一次发现供应商内核的问题，请检查已知最新版本的普通构建是否可以正常运行。
- 向 [Linux 稳定版邮件列表](mailto:stable@vger.kernel.org) 发送一个简短的问题报告 (stable@vger.kernel.org)。大致描述问题，并解释如何复现。讲清楚首个出现问题的版本和最后一个工作正常的版本。然后等待进一步的指示。

下面的参考章节部分详细解释了这些步骤中的每一步。

### 报告只发生在较旧内核版本线的问题

若您尝试了上述的最新主线内核，但未能在那里复现问题，那么本小节适用于您；以下流程有助于使问题在仍然支持的稳定版或长期支持版本线，或者定期基于最新稳定版或长期支持内核的供应商内核中得到修复。如果是这种情况，请执行以下步骤：

- 请做好准备，接下来的几个步骤可能无法在旧版本中解决问题：修复可能太大或太冒险，无法移植到那里。
- 执行前节“报告稳定版和长期支持内核线的回归”中的前三个步骤。
- 在 Linux 内核版本控制系统中搜索修复主线问题的更改，因为它的提交消息可能会告诉你修复是否已经计划好了支持。如果你没有找到，搜索适当的邮件列表，寻找讨论此类问题或同行评议可能修复的帖子；然后检查讨论是否认为修复不适合支持。如果支持根本不被考虑，加入最新的讨论，询问是否有可能。
- 前面的步骤之一应该会给出一个解决方案。如果仍未能成功，请向可能引起问题的子系统的维护人员询问建议；抄送特定子系统的邮件列表以及稳定版邮件列表

下面的参考章节部分详细解释了这些步骤中的每一步。

### 参考章节：向内核维护者报告问题

上面的详细指南简要地列出了所有主要步骤，这对大多数人来说应该足够了。但有时，即使是有经验的用户也可能想知道如何实际执行这些步骤之一。这就是本节的目的，因为它将提供关于上述每个步骤的更多细节。请将此作为参考文档：可以从头到尾阅读它。但它主要是为了浏览和查找如何实际执行这些步骤的详细信息。

在深入挖掘细节之前，我想先给你一些一般性建议：

- Linux 内核开发人员很清楚这个过程很复杂，比其他的 FLOSS 项目要求更多。我们很希望让它更简单。但这需要在不同的地方以及一些基础设施上付诸努力，这些基础设施需要持续的维护；尚未有人站出来做这些工作，所以目前情况就是这样。
- 与某些供应商签订的保证或支持合同并不能使您有权要求上游 Linux 内核社区的开发人员进行修复：这样的合同完全在 Linux 内核、其开发社区和本文档的范围之外。这就是为什么在这种情况下，你不能要求任何契约保证，即使开发人员处理的问题对供应商有效。如果您想主张您的权利，使用供应商的支持渠道代替。当这样做的时候，你可能想提出你希望看到这个问题在上游 Linux 内核中修复；可以这是确保最终修复将被纳入所有 Linux 发行版的唯一方法来鼓励他们。
- 如果您从未向 FLOSS 项目报告过任何问题，那么您应该考虑阅读 [如何有效地报告缺陷](#)，[如何以明智的方式提问](#)，和 [如何提出好问题](#)。

解决这些问题之后，可以在下面找到如何正确地向 Linux 内核报告问题的详细信息。

## 确保您使用的是上游 Linux 内核

您是否面临硬件或软件供应商提供的 Linux 内核的问题？那么基本上您最好停止阅读本文档，转而向您的供应商报告问题，除非您愿意自己安装最新的 Linux 版本。寻找和解决问题往往需要后者。

与大多数程序员一样，Linux 内核开发人员不喜欢花时间处理他们维护的源代码中根本不会发生的问题的报告。这只会浪费每个人的时间，尤其是你的时间。不幸的是，当涉及到内核时，这样的情况很容易发生，并且常常导致双方气馁。这是因为几乎所有预装在设备（台式机、笔记本电脑、智能手机、路由器等）上的 Linux 内核，以及大多数由 Linux 发行商提供的内核，都与由 kernel.org 发行的官方 Linux 内核相距甚远：从 Linux 开发的角度来看，这些供应商提供的内核通常是古老的或者经过了大量修改，通常两点兼具。

大多数供应商内核都不适合用来向 Linux 内核开发人员报告问题：您在其中遇到的问题可能已经由 Linux 内核开发人员在数月或数年前修复；此外，供应商的修改和增强可能会导致您面临的问题，即使它们看起来很小或者完全不相关。这就是为什么您应该向供应商报告这些内核的问题。它的开发者应该查看报告，如果它是一个上游问题，直接于上游修复或将报告转发到那里。在实践中，这有时行不通。因此，您可能需要考虑通过自己安装最新的 Linux 内核内核来绕过供应商。如果如果您选择此方法，那么本指南后面的步骤将解释如何在排除了其他可能导致您的问题的原因后执行此操作。

注意前段使用的词语是“大多数”，因为有时候开发人员实际上愿意处理供应商内核出现的问题报告。他们是否这么做很大程度上取决于开发人员和相关问题。如果发行版只根据最近的 Linux 版本对内核进行了较小修改，那么机会就比较大；例如对于 Debian GNU/Linux Sid 或 Fedora Rawhide 所提供的主线内核。一些开发人员还将接受基于最新稳定内核的发行版内核问题报告，只要它改动不大；例如 Arch Linux、常规 Fedora 版本和 openSUSE Turboweed。但是请记住，您最好使用主线 Linux，并避免在此流程中使用稳定版内核，如“安装一个新的内核进行测试”一节中所详述。

当然，您可以忽略所有这些建议，并向上游 Linux 开发人员报告旧的或经过大量修改的供应商内核的问题。但是注意，这样的报告经常被拒绝或忽视，所以自行小心考虑一下。不过这还是比根本不报告问题要好：有时候这样的报告会直接或间接地帮助解决之后的问题。

## 搜索现有报告（第一部分）

使用您喜爱的网络搜索引擎对现有报告进行粗略搜索；此外，请检查 Linux 内核邮件列表 (LKML) 的存档。如果找到匹配的报告，请加入讨论而不是发送新报告。

报告一个别人已经提出的问题，对每个人来说都是浪费时间，尤其是作为报告人的你。所以彻底检查是否有人已经报告了这个问题，这对你自己是有利的。在流程中的这一步，可以只执行一个粗略的搜索：一旦您知道您的问题需要报告到哪里，稍后的步骤将告诉您如何详细搜索。尽管如此，不要仓促完成这一步，它可以节省您的时间和减少麻烦。

只需先用你最喜欢的搜索引擎在互联网上搜索。然后再搜索 Linux 内核邮件列表 (LKML) 存档。

如果搜索结果实在太多，可以考虑让你的搜索引擎将搜索时间范围限制在过去的一个月或一年。而且无论你在哪里搜索，一定要用恰当的搜索关键词；也要变化几次关键词。同时，试着从别人的角度看问题：这将帮助你想出其他的关键词。另外，一定不要同时使用过多的关键词。记住搜索时要同时尝试包含和不包含内核驱动程序的名称或受影响的硬件组件的名称等信息。但其确切的品牌名称（比如说“华硕红魔 Radeon RX 5700 XT

Gaming OC”) 往往帮助不大，因为它太具体了。相反，尝试搜索术语，如型号（Radeon 5700 或 Radeon 5000）和核心代号（“Navi” 或 “Navi10”），以及包含和不包含其制造商（“AMD”）。

如果你发现了关于你的问题的现有报告，请加入讨论，因为你可能会提供有价值的额外信息。这一点很重要，即使是在修复程序已经准备好或处于最后阶段，因为开发人员可能会寻找能够提供额外信息或测试建议修复程序的人。跳到“发布报告后的责任”一节，了解有关如何正确参与的细节。

注意，搜索 [bugzilla.kernel.org](https://bugzilla.kernel.org) 网站可能也是一个好主意，因为这可能会提供有价值的见解或找到匹配的报告。如果您发现后者，请记住：大多数子系统都希望在不同的位置报告，如下面“你需要将问题报告到何处”一节中所述。因此本应处理这个问题的开发人员甚至可能不知道 bugzilla 的工单。所以请检查工单中的问题是否已经按照本文档所述得到报告，如果没有，请考虑这样做。

## 高优先级的问题？

看看你正在处理的问题是否是回归问题、安全问题或非常严重的问题：这些都是需要在接下来的一些步骤中特别处理的“高优先级问题”。

Linus Torvalds 和主要的 Linux 内核开发人员希望看到一些问题尽快得到解决，因此在报告过程中有一些“高优先级问题”的处理略有不同。有三种情况符合条件：回归、安全问题和非常严重的问题。

如果在旧版本的 Linux 内核中工作的东西不能在新版本的 Linux 内核中工作，或者某种程度上在新版本的 Linux 内核中工作得更差，那么你就需要处理“回归”。因此，当一个在 Linux 5.7 中表现良好的 WiFi 驱动程序在 5.8 中表现不佳或根本不能工作时，这是一种回归。如果应用程序在新的内核中出现不稳定的现象，这也是一种回归，这可能是由于内核和用户空间之间的接口（如 procfs 和 sysfs）发生不兼容的更改造成的。显著的性能降低或功耗增加也可以称为回归。但是请记住：新内核需要使用与旧内核相似的配置来构建（参见下面如何实现这一点）。这是因为内核开发人员在实现新特性时有时无法避免不兼容性；但是为了避免回归，这些特性必须在构建配置期间显式地启用。

什么是安全问题留给自己判断。在继续之前，请考虑阅读“安全缺陷”，因为它提供了如何最恰当地处理安全问题的额外细节。

当发生了完全无法接受的糟糕事情时，此问题就是一个“非常严重的问题”。例如，Linux 内核破坏了它处理的数据或损坏了它运行的硬件。当内核突然显示错误消息（“kernel panic”）并停止工作，或者根本没有任何停止信息时，您也在处理一个严重的问题。注意：不要混淆“panic”（内核停止自身的致命错误）和“Oops”（可恢复错误），因为显示后者之后内核仍然在运行。

## 确保环境健康

确保不是内核所处环境导致了你所面临的问题。

看起来很像内核问题的问题有时是由构建或运行时环境引起的。很难完全排除这种问题，但你应该尽量减少这种问题：

- 构建内核时，请使用经过验证的工具，因为编译器或二进制文件中的错误可能会导致内核出现错误行为。
- 确保您的计算机组件在其设计规范内运行；这对处理器、内存和主板尤为重要。因此，当面临潜在的内核问题时，停止低电压或超频。

- 尽量确保不是硬件故障导致了你的问题。例如，内存损坏会导致大量的问题，这些问题会表现为看起来像内核问题。
- 如果你正在处理一个文件系统问题，你可能需要用 `fsck` 检查一下文件系统，因为它可能会以某种方式被损坏，从而导致无法预期的内核行为。
- 在处理回归问题时，要确保没有在更新内核的同时发生了其他变化。例如，这个问题可能是由同时更新的其他软件引起的。也有可能是在你第一次重启进入新内核时，某个硬件巧合地坏了。更新系统 BIOS 或改变 BIOS 设置中的某些内容也会导致一些看起来很像内核回归的问题。

## 为紧急情况做好准备

创建一个全新的备份，并将系统修复和还原工具放在手边

我得提醒您，您正在和计算机打交道，计算机有时会出现意想不到的事情，尤其是当您折腾其操作系统的内核等关键部件时。而这就是你在这个过程中要做的事情。因此，一定要创建一个全新的备份；还要确保你手头有修复或重装操作系统的所有工具，以及恢复备份所需的一切。

## 确保你的内核不会被增强

确保您的系统不会通过动态构建额外的内核模块来增强其内核，像 `DKMS` 这样的解决方案可能在您不知情的情况下就在本地进行了这样的工作。

如果内核以任何方式得到增强，那么问题报告被忽略或拒绝的风险就会急剧增加。这就是为什么您应该删除或禁用像 `akmods` 和 `DKMS` 这样的机制：这些机制会自动构建额外内核模块，例如当您安装新的 Linux 内核或第一次引导它时。也要记得同时删除他们可能安装的任何模块。然后重新启动再继续。

注意，你可能不知道你的系统正在使用这些解决方案之一：当你安装 Nvidia 专有图形驱动程序、VirtualBox 或其他需要 Linux 内核以外的模块支持的软件时，它们通常会静默设置。这就是为什么你可能需要卸载这些软件的软件包，以摆脱任何第三方内核模块。

## 检查“污染”标志

当问题发生时，检查您的内核是否被“污染”，因为使内核设置这个标志的事件可能会导致您面临的问题。

当某些可能会导致看起来完全不相关的后续错误的事情发生时，内核会用“污染（taint）”标志标记自己。如果您的内核受到污染，那么您面临的可能是这样的错误。因此在投入更多时间到这个过程中之前，尽早排除此情况可能对你有好处。这是这个步骤出现在这里的唯一原因，因为这个过程稍后会告诉您安装最新的主线内核；然后您将需要再次检查污染标志，因为当它出问题的时候内核报告会关注它。

在正在运行的系统上检查内核是否污染非常容易：如果 `cat /proc/sys/kernel/tainted` 返回“0”，那么内核没有被污染，一切正常。在某些情况下无法检查该文件；这就是为什么当内核报告内部问题（“kernel bug”）、可恢复错误（“kernel Oops”）或停止操作前不可恢复的错误（“kernel panic”）时，它也会提到污染状态。

当其中一个错误发生时，查看打印的错误消息的顶部，搜索以“CPU:”开头的行。如果发现问题时内核未被污染，那么它应该以“Not infected”结束；如果你看到“Tainted:”且后跟一些空格和字母，那就被污染了。如果你的内核被污染了，请阅读“[受污染的内核](#)”以找出原因。设法消除污染因素。通常是由以下三种因素之一引起的：

1. 发生了一个可恢复的错误（“kernel Oops”），内核污染了自己，因为内核知道在此之后它可能会出现奇怪的行为错乱。在这种情况下，检查您的内核或系统日志，并寻找以下列文字开头的部分：

```
Oops: 0000 [#1] SMP
```

如方括号中的“#1”所示，这是自启动以来的第一次 Oops。每个 Oops 和此后发生的任何其他问题都可能是首个 Oops 的后续问题，即使这两个问题看起来完全不相关。通过消除首个 Oops 的原因并在之后复现该问题，可以排除这种情况。有时仅仅重新启动就足够了，有时更改配置后重新启动可以消除 Oops。但是在这个流程中不要花费太多时间在这一点上，因为引起 Oops 的原因可能已经在您稍后将按流程安装的新 Linux 内核版本中修复了。

2. 您的系统使用的软件安装了自己的内核模块，例如 Nvidia 的专有图形驱动程序或 VirtualBox。当内核从外部源（即使它们是开源的）加载此类模块时，它会污染自己：它们有时会在不相关的内核区域导致错误，从而可能导致您面临的问题。因此，当您想要向 Linux 内核开发人员报告问题时，您必须阻止这些模块加载。大多数情况下最简单的方法是：临时卸载这些软件，包括它们可能已经安装的任何模块。之后重新启动。
3. 当内核加载驻留在 Linux 内核源代码 staging 树中的模块时，它也会污染自身。这是一个特殊的区域，代码（主要是驱动程序）还没有达到正常 Linux 内核的质量标准。当您报告此种模块的问题时，内核受到污染显然是没有问题的；只需确保问题模块是造成污染的唯一原因。如果问题发生在一个不相关的区域，重新启动并通过指定 `foo.blacklist=1` 作为内核参数临时阻止该模块被加载（用有问题的模块名替换“foo”）。

### 记录如何重现问题

粗略地写下如何重现这个问题。如果您同时处理多个问题，请为每个问题单独写注释，并确保它们在新启动的系统上独立出现。这是必要的，因为每个问题都需要分别报告给内核开发人员，除非它们严重纠缠在一起。

如果你同时处理多个问题，必须分别报告每个问题，因为它们可能由不同的开发人员处理。在一份报告中描述多种问题，也会让其他人难以将其分开。因此只有在问题严重纠缠的情况下，才能将问题合并到一份报告中。

此外，在报告过程中，你必须测试该问题是否发生在其他内核版本上。因此，如果您知道如何在一个新启动的系统上快速重现问题，将使您的工作更加轻松。

注意：报告只发生过一次的问题往往是没有结果的，因为它们可能是由于宇宙辐射导致的位翻转。所以你应该尝试通过重现问题来排除这种情况，然后再继续。如果你有足够的经验来区分由于硬件故障引起的一次性错误和难以重现的罕见内核问题，可以忽略这个建议。

## 稳定版或长期支持内核的回归?

如果您正面临稳定版或长期支持版本线的回归（例如从 5.10.4 更新到 5.10.5 时出现故障），请查看后文“报告稳定版和长期支持内核线的回归”小节。

稳定版和长期支持内核版本线中的回归是 Linux 开发人员非常希望解决的问题，这样的问题甚至比主线开发分支中的回归更不应出现，因为它们会很快影响到很多人。开发人员希望尽快了解此类问题，因此有一个简化流程来报告这些问题。注意，使用更新内核版本线的回归（比如从 5.9.15 切换到 5.10.5 时出现故障）不符合条件。

## 你需要将问题报告到何处

定位可能引起问题的驱动程序或内核子系统。找出其开发人员期望的报告的方式和位置。注意：大多数情况下不会是 [bugzilla.kernel.org](https://bugzilla.kernel.org)，因为问题通常需要通过邮件发送给维护人员和公共邮件列表。

将报告发送给合适的人是至关重要的，因为 Linux 内核是一个大项目，大多数开发人员只熟悉其中的一小部分。例如，相当多的程序员只关心一个驱动程序，比如一个 WiFi 芯片驱动程序；它的开发人员可能对疏远的或不相关的“子系统”（如 TCP 堆栈、PCIe/PCI 子系统、内存管理或文件系统）的内部知识了解很少或完全不了解。

问题在于：Linux 内核缺少一个，可以简单地将问题归档并让需要了解它的开发人员了解它的，中心化缺陷跟踪器。这就是为什么你必须找到正确的途径来自己报告问题。您可以在脚本的帮助下做到这一点（见下文），但它主要针对的是内核开发人员和专家。对于其他人来说，MAINTAINERS（维护人员）文件是更好的选择。

## 如何阅读 MAINTAINERS 维护者文件

为了说明如何使用 MAINTAINERS 文件，让我们假设您的笔记本电脑中的 WiFi 在更新内核后突然出现了错误行为。这种情况下可能是 WiFi 驱动的问题。显然，它也可能由于驱动基于的某些代码，但除非你怀疑有这样东西会附着在驱动程序上。如果真的是其他的问题，驱动程序的开发人员会让合适的人参与进来。

遗憾的是，没有通用且简单的办法来检查哪个代码驱动了特定硬件组件。

在 WiFi 驱动出现问题的情况下，你可能想查看 `lspci -k` 的输出，因为它列出了 PCI/PCIe 总线上的设备和驱动它的内核模块：

```
[user@something ~]$ lspci -k
[...]
3a:00.0 Network controller: Qualcomm Atheros QCA6174 802.11ac Wireless Network Adapter (rev 32)
    Subsystem: Bigfoot Networks, Inc. Device 1535
    Kernel driver in use: ath10k_pci
    Kernel modules: ath10k_pci
[...]
```

但如果您的 WiFi 芯片通过 USB 或其他内部总线连接，这种方法就行不通了。在这种情况下，您可能需要检查您的 WiFi 管理器或 `ip link` 的输出。寻找有问题的网络接口的名称，它可能类似于“wlp58s0”。此名称

可以用来找到驱动它的模块:

```
[user@something ~]$ realpath --relative-to=/sys/module//sys/class/net/wlp58s0/device/driver/  
└─module  
ath10k_pci
```

如果这些技巧不能进一步帮助您, 请尝试在网上搜索如何缩小相关驱动程序或子系统的范围。如果你不确定是哪一个: 试着猜一下, 即使你猜得不好, 也会有人会帮助你的。

一旦您知道了相应的驱动程序或子系统, 您就希望在 MAINTAINERS 文件中搜索它。如果是 “ath10k\_pci”, 您不会找到任何东西, 因为名称太具体了。有时你需要在网上寻找帮助; 但在此之前, 请尝试使用一个稍短或修改过的名称来搜索 MAINTAINERS 文件, 因为这样你可能会发现类似这样的东西:

QUALCOMM AHEROS ATH10K WIRELESS DRIVER	
Mail:	A. Some Human < <a href="mailto:shuman@example.com">shuman@example.com</a> >
Mailing list:	<a href="mailto:ath10k@lists.infradead.org">ath10k@lists.infradead.org</a>
Status:	Supported
Web-page:	<a href="https://wireless.wiki.kernel.org/en/users/Drivers/ath10k">https://wireless.wiki.kernel.org/en/users/Drivers/ath10k</a>
SCM:	<a href="git://git.kernel.org/pub/scm/linux/kernel/git/kvalo/ath.git">git git://git.kernel.org/pub/scm/linux/kernel/git/kvalo/ath.git</a>
Files:	<a href="#">drivers/net/wireless/ath/ath10k/</a>

注意: 如果您阅读在 Linux 源代码树的根目录中找到的原始维护者文件, 则行描述将是缩写。例如, “Mail: (邮件)” 将是 “M:”, “Mailing list: (邮件列表)” 将是 “L”, “Status: (状态)” 将是 “S:”。此文件顶部有一段解释了这些和其他缩写。

首先查看 “Status” 状态行。理想情况下, 它应该得到 “Supported (支持)” 或 “Maintained (维护)”。如果状态为 “Obsolete (过时的)”, 那么你在使用一些过时的方法, 需要转换到新的解决方案上。有时候, 只有在感到有动力时, 才会有人为代码提供 “Odd Fixes”。如果碰见 “Orphan”, 你就完全不走运了, 因为再也没有人关心代码了, 只剩下这些选项: 准备好与问题共存, 自己修复它, 或者找一个愿意修复它的程序员。

检查状态后, 寻找以 “bug:” 开头的一行: 它将告诉你在哪里可以找到子系统特定的缺陷跟踪器来提交你的问题。上面的例子没有此行。大多数部分都是这样, 因为 Linux 内核的开发完全是由邮件驱动的。很少有子系统使用缺陷跟踪器, 且其中只有一部分依赖于 [bugzilla.kernel.org](https://bugzilla.kernel.org)。

在这种以及其他很多情况下, 你必须寻找以 “Mail:” 开头的行。这些行提到了特定代码的维护者的名字和电子邮件地址。也可以查找以 “Mailing list:” 开头的行, 它告诉你开发代码的公共邮件列表。你的报告之后需要通过邮件发到这些地址。另外, 对于所有通过电子邮件发送的问题报告, 一定要抄送 Linux Kernel Mailing List (LKML) <[linux-kernel@vger.kernel.org](mailto:linux-kernel@vger.kernel.org)>。在以后通过邮件发送问题报告时, 不要遗漏任何一个邮件列表! 维护者都是大忙人, 可能会把一些工作留给子系统特定列表上的其他开发者; 而 LKML 很重要, 因为需要一个可以找到所有问题报告的地方。

## 借助脚本找到维护者

对于手头有 Linux 源码的人来说，有第二个可以找到合适的报告地点的选择：脚本“scripts/get\_maintainer.pl”，它尝试找到所有要联系的人。它会查询 MAINTAINERS 文件，并需要用相关源代码的路径来调用。对于编译成模块的驱动程序，经常可以用这样的命令找到：

```
$ modinfo ath10k_pci | grep filename | sed 's!/lib/modules/.*/kernel/!!; s!filename:!!; s!\.ko\
˓→(\|\.xz\)\!!'
drivers/net/wireless/ath/ath10k/ath10k_pci.ko
```

将其中的部分内容传递给脚本：

```
$ ./scripts/get_maintainer.pl -f drivers/net/wireless/ath/ath10k*
Some Human <shuman@example.com> (supporter:QUALCOMM Atheros ATH10K WIRELESS DRIVER)
Another S. Human <asomehuman@example.com> (maintainer:NETWORKING DRIVERS)
ath10k@lists.infradead.org (open list:QUALCOMM Atheros ATH10K WIRELESS DRIVER)
linux-wireless@vger.kernel.org (open list:NETWORKING DRIVERS (WIRELESS))
netdev@vger.kernel.org (open list:NETWORKING DRIVERS)
linux-kernel@vger.kernel.org (open list)
```

不要把你的报告发给所有的人。发送给维护者，脚本称之为“supporter”；另外抄送代码最相关的邮件列表，以及 Linux 内核邮件列表（LKML）。在此例中，你需要将报告发送给“Some Human <shuman@example.com>”，并抄送“ath10k@lists.infradead.org”和“linux-kernel@vger.kernel.org”。

注意：如果你用 git 克隆了 Linux 源代码，你可能需要用–git 再次调用 get\_maintainer.pl。脚本会查看提交历史，以找到最近哪些人参与了相关代码的编写，因为他们可能会提供帮助。但要小心使用这些结果，因为它很容易让你误入歧途。例如，这种情况常常会在很少被修改的地方（比如老旧的或未维护的驱动程序）：有时这样的代码会在树级清理期间被根本不关心此驱动程序的开发者修改。

## 搜索现有报告（第二部分）

在缺陷追踪器或问题相关邮件列表的存档中彻底搜索可能与您的问题匹配的报告。如果找到匹配的报告，请加入讨论而不是发送新报告。

如前所述：报告一个别人已经提出的问题，对每个人来说都是浪费时间，尤其是作为报告人的你。这就是为什么你应该再次搜索现有的报告。现在你已经知道问题需要报告到哪里。如果是邮件列表，那么一般在 [lore.kernel.org](#) 可以找到相应存档。

但有些列表运行在其他地方。例如前面步骤中当例子的 ath10k WiFi 驱动程序就是这种情况。但是你通常可以在网上很容易地找到这些列表的档案。例如搜索“archive [ath10k@lists.infradead.org](#)”，将引导您到 ath10k 邮件列表的信息页，该页面顶部链接到其 [列表存档](#)。遗憾的是，这个列表和其他一些列表缺乏搜索其存档的功能。在这种情况下可以使用常规的互联网搜索引擎，并添加类似“site:[lists.infradead.org/pipermail/ath10k/](#)”这样的搜索条件，这会把结果限制在该链接中的档案。

也请进一步搜索网络、LKML 和 bugzilla.kernel.org 网站。

有关如何搜索以及在找到匹配报告时如何操作的详细信息，请参阅上面的“[搜索现有报告（第一部分）](#)”。

不要急着完成报告过程的这一步：花 30 到 60 分钟甚至更多的时间可以为你和其他人节省 / 减少相当多的时间和麻烦。

### 安装一个新的内核进行测试

除非您已经在运行最新的“主线”Linux 内核，否则最好在报告流程前安装它。在某些情况下，使用最新的“稳定版”Linux 进行测试和报告也是可以接受的替代方案；在合并窗口期间，这实际上可能是最好的方法，但在开发阶段最好还是暂停几天。无论你选择什么版本，最好使用“普通”构建。忽略这些建议会大大增加您的报告被拒绝或忽略的风险。

正如第一步的详细解释中所提到的：与大多数程序员一样，与大多数程序员一样，Linux 内核开发人员不喜欢花时间处理他们维护的源代码中根本不会发生的问题的报告。这只会浪费每个人的时间，尤其是你的时间。这就是为什么在报告问题之前，您必须先确认问题仍然存在于最新的上游代码中，这符合每个人的利益。您可以忽略此建议，但如前所述：这样做会极大地增加问题报告被拒绝或被忽略的风险。

内核“最新上游”的范围通常指：

- 安装一个主线内核；最新的稳定版内核也可以是一个选择，但大多数时候都最好避免。长期支持内核（有时称为“LTS 内核”）不适合此流程。下一小节将更详细地解释所有这些。
- 下一小节描述获取和安装这样一个内核的方法。它还指出了使用预编译内核是可以的，但普通的内核更好，这意味着：它是直接使用从 [kernel.org](https://kernel.org) 获得的 Linux 源代码构建并且没有任何方式修改或增强。

### 选择适合测试的版本

前往 [kernel.org](https://kernel.org) 来决定使用哪个版本。忽略那个写着“Latest release 最新版本”的巨大黄色按钮，往下看有一个表格。在表格的顶部，你会看到一行以“mainline”开头的字样，大多数情况下它会指向一个版本号类似“5.8-rc2”的预发布版本。如果是这样的话，你将需要使用这个主线内核进行测试。不要让“rc”吓到你，这些“开发版内核”实际上非常可靠——而且你已经按照上面的指示做了备份，不是吗？

大概每九到十周，“mainline”可能会给你指出一个版本号类似“5.7”的正式版本。如果碰见这种情况，请考虑暂停报告过程，直到下一个版本的第一个预发布（5.8-rc1）出现在 [kernel.org](https://kernel.org) 上。这是因为 Linux 的开发周期正在两周的“合并窗口”内。大部分的改动和所有干扰性的改动都会在这段时间内被合并到下一个版本中。在此期间使用主线是比较危险的。内核开发者通常也很忙，可能没有多余的时间来处理问题报告。这也是很有可能在合并窗口中应用了许多修改来修复你所面临的问题；这就是为什么你很快就得用一个新的内核版本重新测试，就像下面“发布报告后的责任”一节中所述的那样。

这就是为什么要等到合并窗口结束后才去做。但是如果你处理的是一些不应该等待的东西，则无需这样做。在这种情况下，可以考虑通过 git 获取最新的主线内核（见下文），或者使用 [kernel.org](https://kernel.org) 上提供的最新稳定版本。如果 mainline 因为某些原因无法正常工作，那么使用它也是可以接受的。总的来说：用它来重现问题也比完全不报告问题要好。

最好避免在合并窗口外使用最新的稳定版内核，因为所有修复都必须首先应用于主线。这就是为什么检查最新的主线内核是如此重要：你希望看到在旧版本线修复的任何问题需要先在主线修复，然后才能得到回传，这可

能需要几天或几周。另一个原因是：您希望的修复对于回传来说可能太难或太冒险；因此再次报告问题不太可能改变任何事情。

这些方面也部分表明了为什么长期支持内核（有时称为“LTS 内核”）不适合报告流程：它们与当前代码的距离太远。因此，先去测试主线，然后再按流程走：如果主线没有出现问题，流程将指导您如何在旧版本线中修复它。

## 如何获得新的 Linux 内核

你可以使用预编译或自编译的内核进行测试；如果你选择后者，可以使用 `git` 获取源代码，或者下载其 tar 存档包。

**使用预编译的内核：**这往往是最快速、最简单、最安全的方法——尤其是在你不熟悉 Linux 内核的情况下。问题是：发行商或附加存储库提供的大多数版本都是从修改过的 Linux 源代码构建的。因此它们不是普通的，通常不适合于测试和问题报告：这些更改可能会导致您面临的问题或以某种方式影响问题。

但是如果您使用的是流行的 Linux 发行版，那么您就很幸运了：对于大部分的发行版，您可以在网上找到包含最新主线或稳定版本 Linux 内核包的存储库。使用这些是完全可以的，只要从存储库的描述中确认它们是普通的或者至少接近普通。此外，请确保软件包包含 [kernel.org](#) 上提供的最新版本内核。如果这些软件包的时间超过一周，那么它们可能就不合适了，因为新的主线和稳定版内核通常至少每周发布一次。

请注意，您以后可能需要手动构建自己的内核：有时这是调试或测试修复程序所必需的，如后文所述。还要注意，预编译的内核可能缺少在出现 `panic`、`Oops`、`warning` 或 `BUG` 时解码内核打印的消息所需的调试符号；如果您计划解码这些消息，最好自己编译内核（有关详细信息，请参阅本小节结尾和“解码失败信息”小节）。

**使用 git：**熟悉 `git` 的开发者和有经验的 Linux 用户通常最好直接从 [kernel.org 上的官方开发仓库](#) 中获取最新的 Linux 内核源代码。这些很可能比最新的主线预发布版本更新一些。不用担心：它们和正式的预发布版本一样可靠，除非内核的开发周期目前正处于合并窗口中。不过即便如此，它们也是相当可靠的。

**常规方法：**不熟悉 `git` 的人通常最好从 [kernel.org](#) 下载源码的 tar 存档包。

如何实际构建一个内核并不在这里描述，因为许多网站已经解释了必要的步骤。如果你是新手，可以考虑按照那些建议使用 `make localmodconfig` 来做，它将尝试获取你当前内核的配置，然后根据你的系统进行一些调整。这样做并不能使编译出来的内核更好，但可以更快地编译。

注意：如果您正在处理来自内核的 `panic`、`Oops`、`warning` 或 `BUG`，请在配置内核时尝试启用 `CONFIG_KALLSYMS` 选项。此外，还可以启用 `CONFIG_DEBUG_KERNEL` 和 `CONFIG_DEBUG_INFO`；后者是相关选项，但只有启用前者才能开启。请注意，`CONFIG_DEBUG_INFO` 会需要更多储存空间来构建内核。但这是值得的，因为这些选项将允许您稍后精确定位触发问题的确切代码行。下面的“解码失败信息”一节对此进行了更详细的解释。

但请记住：始终记录遇到的问题，以防难以重现。发送未解码的报告总比不报告要好。

### 检查“污染”标志

确保您刚刚安装的内核在运行时不会“污染”自己。

正如上面已经详细介绍过的：当发生一些可能会导致一些看起来完全不相关的后续错误的事情时，内核会设置一个“污染”标志。这就是为什么你需要检查你刚刚安装的内核是否有设置此标志。如果有的话，几乎在任何情况下你都需要在报告问题之前先消除它。详细的操作方法请看上面的章节。

### 用新内核重现问题

在您刚刚安装的内核中复现这个问题。如果它没有出现，请查看下方只发生在稳定版和长期支持内核的问题的说明。

检查这个问题是否发生在你刚刚安装的新 Linux 内核版本上。如果新内核已经修复了，可以考虑使用此版本线，放弃报告问题。但是请记住，只要它没有在 [kernel.org](#) 的稳定版和长期版（以及由这些版本衍生出来的厂商内核）中得到修复，其他用户可能仍然会受到它的困扰。如果你喜欢使用其中的一个，或者只是想帮助它们的用户，请前往下面的“报告只发生在较旧内核版本线的问题”一节。

### 优化复现问题的描述

优化你的笔记：试着找到并写出最直接的复现问题的方法。确保最终结果包含所有重要的细节，同时让第一次听说的人容易阅读和理解。如果您在此过程中学到了一些东西，请考虑再次搜索关于该问题的现有报告。

过于复杂的报告会让别人很难理解。因此请尽量找到一个可以直接描述、易于以书面形式理解的再现方法。包含所有重要的细节，但同时也要尽量保持简短。

在这在前面的步骤中，你很可能已经了解了一些关于你所面临的问题的点。利用这些知识，再次搜索可以转而加入的现有报告。

### 解码失败信息

如果失败涉及“panic”、“Oops”、“warning”或“BUG”，请考虑解码内核日志以查找触发错误的代码行。

当内核检测到内部问题时，它会记录一些有关已执行代码的信息。这使得在源代码中精确定位触发问题的行并显示如何调用它成为可能。但只有在配置内核时启用了 CONFIG\_DEBUG\_INFO 和 CONFIG\_KALLSYMS 选项时，这种方法才起效。如果已启用此选项，请考虑解码内核日志中的信息。这将使我们更容易理解是什么导致了“panic”、“Oops”、“warning”或“BUG”，从而增加了有人提供修复的几率。

解码可以通过 Linux 源代码树中的脚本来完成。如果您运行的内核是之前自己编译的，这样这样调用它：

```
[user@something ~]$ sudo dmesg | ./linux-5.10.5/scripts/decode_stacktrace.sh ./linux-5.10.5/
↳ vmlinux
/usr/lib/debug/lib/modules/5.10.10-4.1.x86_64/vmlinux /usr/src/kernels/5.10.10-4.1.x86_64/
```

如果您运行的是打包好的普通内核，则可能需要安装带有调试符号的相应包。然后按以下方式调用脚本（如果发行版未打包，则可能需要从 Linux 源代码获取）：

```
[user@something ~]$ sudo dmesg | ./linux-5.10.5/scripts/decode_stacktrace.sh \
/usr/lib/debug/lib/modules/5.10.10-4.1.x86_64/vmlinux /usr/src/kernels/5.10.10-4.1.x86_64/
```

脚本将解码如下的日志行，这些日志行显示内核在发生错误时正在执行的代码的地址：

```
[ 68.387301] RIP: 0010:test_module_init+0x5/0xffa [test_module]
```

解码之后，这些行将变成这样：

```
[ 68.387301] RIP: 0010:test_module_init (/home/username/linux-5.10.5/test-module/test-module.
↪c:16) test_module
```

在本例中，执行的代码是从文件“`~/linux-5.10.5/test-module/test-module.c`”构建的，错误出现在第 16 行的指令中。

该脚本也会如此解码以“Call trace”开头的部分中提到的地址，该部分显示出问题的函数的路径。此外，脚本还会显示内核正在执行的代码部分的汇编输出。

注意，如果你没法做到这一点，只需跳过这一步，并在报告中说明原因。如果你幸运的话，可能无需解码。如果需要的话，也许有人会帮你做这件事情。还要注意，这只是解码内核堆栈跟踪的几种方法之一。有时需要采取不同的步骤来检索相关的详细信息。别担心，如果您碰到的情况需要这样做，开发人员会告诉您该怎么做。

## 对回归的特别关照

如果您的问题是回归问题，请尽可能缩小引入问题时的范围。

Linux 首席开发者 Linus Torvalds 认为 Linux 内核永远不应恶化，这就是为什么他认为回归是不可接受的，并希望看到它们被迅速修复。这就是为什么引入了回归的改动导致的问题若无法通过其他方式快速解决，通常会被迅速撤销。因此，报告回归有点像“王炸”，会迅速得到修复。但要做到这一点，需要知道导致回归的变化。通常情况下，要由报告者来追查罪魁祸首，因为维护者往往没有时间或手头设置不便来自行重现它。

有一个叫做“二分”的过程可以来寻找变化，这在“[二分 \(bisect\) 缺陷](#)”文档中进行了详细的描述，这个过程通常需要你构建十到二十个内核镜像，每次都尝试在构建下一个镜像之前重现问题。是的，这需要花费一些时间，但不用担心，它比大多数人想象的要快得多。多亏了“binary search 二进制搜索”，这将引导你在源代码管理系统中找到导致回归的提交。一旦你找到它，就在网上搜索其主题、提交 ID 和缩短的提交 ID（提交 ID 的前 12 个字符）。如果说有的话，这将引导您找到关于它的现有报告。

需要注意的是，二分法需要一点窍门，不是每个人都懂得诀窍，也需要相当多的努力，不是每个人都愿意投入。尽管如此，还是强烈建议自己进行一次二分。如果你真的不能或者不想走这条路，至少要找出是哪个主线内核引入的回归。比如说从 5.5.15 切换到 5.8.4 的时候出现了一些问题，那么至少可以尝试一下相近的所有主线版本（5.6、5.7 和 5.8）来检查它是什么时候出现的。除非你想在一个稳定版或长期支持内核中找到一个回归，否则要避免测试那些编号有三段的版本（5.6.12、5.7.8），因为那会使结果难以解释，可能会让你的测试变得无用。一旦你找到了引入回归的主要版本，就可以放心地继续报告了。但请记住：在不知道罪魁祸首的情

况下，开发人员是否能够提供帮助取决于手头的问题。有时他们可能会从报告中确认是什么出现了问题，并能修复它；有时他们可能无法提供帮助，除非你进行二分。

当处理回归问题时，请确保你所面临的问题真的是由内核引起的，而不是由其他东西引起的，如上文所述。

在整个过程中，请记住：只有当旧内核和新内核的配置相似时，问题才算回归。最好的方法是：把配置文件 (`.config`) 从旧的工作内核直接复制到你尝试的每个新内核版本。之后运行 `make oldnoconfig` 来调整它以适应新版本的需要，而不启用任何新的功能，因为那些功能也可能导致回归。

### 撰写并发送报告

通过详细描述问题来开始编写报告。记得包括以下条目：您为复现而安装的最新内核版本、使用的 Linux 发行版以及关于如何复现该问题的说明。如果可能，将内核构建配置 (`.config`) 和 ```dmesg``` 的输出放在网上的某个地方，并链接到它。包含或上传所有其他可能相关的信息，如 `Oops` 的输出/截图或来自 ```lspci``` 的输出。一旦你写完了这个主要部分，请在上方插入一个正常长度的段落快速概述问题和影响。再在此之上添加一个简单描述问题的句子，以得到人们的阅读。现在给出一个更短的描述性标题或主题。然后就可以像 `MAINTAINERS` 文件告诉你的那样发送或提交报告了，除非你在处理一个“高优先级问题”：它们需要按照下面“高优先级问题的特殊处理”所述特别关照。

现在你已经准备好了一切，是时候写你的报告了。上文前言中链接的三篇文档对如何写报告做了部分解释。这就是为什么本文将只提到一些基本的内容以及 Linux 内核特有的东西。

有一点是符合这两类的：你的报告中最关键的部分是标题/主题、第一句话和第一段。开发者经常会收到许多邮件。因此，他们往往只是花几秒钟的时间浏览一下邮件，然后再决定继续下一封或仔细查看。因此，你报告的开头越好，有人研究并帮助你的机会就越大。这就是为什么你应该暂时忽略他们，先写出详细的报告。;-)

### 每份报告都应提及的事项

详细描述你的问题是如何发生在你安装的新纯净内核上的。试着包含你之前写的和优化过的分步说明，概述你和其他人如何重现这个问题；在极少数无法重现的情况下，尽量描述你做了什么来触发它。

还应包括其他人为了解该问题及其环境而可能需要的所有相关信息。实际需要的东西在很大程度上取决于具体问题，但有些事项你总是应该包括在内：

- `cat /proc/version` 的输出，其中包含 Linux 内核版本号和构建时的编译器。
- 机器正在运行的 Linux 发行版 (`hostnamectl | grep "Operating System"`)
- CPU 和操作系统的架构 (`uname -mi`)
- 如果您正在处理回归，并进行了二分，请提及导致回归的变更的主题和提交 ID。

许多情况下，让读你报告的人多了解两件事也是明智之举：

- 用于构建 Linux 内核的配置 (“`.config`” 文件)
- 内核的信息，你从 `dmesg` 得到的信息写到一个文件里。确保它以像 “Linux version 5.8-1 (`foobar@example.com`) (`gcc (GCC) 10.2.1, GNU ld version 2.34`) #1 SMP Mon Aug 3 14:54:37

UTC 2020”这样的行开始，如果没有，那么第一次启动阶段的重要信息已经被丢弃了。在这种情况下，可以考虑使用 `journalctl -b 0 -k`；或者你也可以重启，重现这个问题，然后调用 `dmesg`。

这两个文件很大，所以直接把它们放到你的报告中是个坏主意。如果你是在缺陷跟踪器中提交问题，那么将它们附加到工单中。如果你通过邮件报告问题，不要用附件附上它们，因为那会使邮件变得太大，可以按下列之一做：

- 将文件上传到某个公开的地方（你的网站，公共文件粘贴服务，在 [bugzilla.kernel.org](#) 上创建的工单……），并在你的报告中放上链接。理想情况下请使用允许这些文件保存很多年的地方，因为它们可能在很多年后对别人有用；例如 5 年或 10 年后，一个开发者正在修改一些代码，而这些代码正是为了修复你的问题。
- 把文件放在一边，然后说明你会在他人回复时再单独发送。只要记得报告发出去后，真正做到这一点就可以了。;-)

## 提供这些东西可能是明智的

根据问题的不同，你可能需要提供更多的背景数据。这里有一些关于提供什么比较好的建议：

- 如果你处理的是内核的“warning”、“OOPS”或“panic”，请包含它。如果你不能复制粘贴它，试着用 `netconsole` 网络终端远程跟踪或者至少拍一张屏幕的照片。
- 如果问题可能与你的电脑硬件有关，请说明你使用的是什么系统。例如，如果你的显卡有问题，请提及它的制造商，显卡的型号，以及使用的芯片。如果是笔记本电脑，请提及它的型号名称，但尽量确保意义明确。例如“戴尔 XPS 13”就不很明确，因为它可能是 2012 年的那款，那款除了看起来和现在销售的没有什么不同之外，两者没有任何共同之处。因此，在这种情况下，要加上准确的型号，例如 2019 年内推出的 XPS 13 型号为“9380”或“7390”。像“联想 Thinkpad T590”这样的名字也有些含糊不清：这款笔记本有带独立显卡和不带的子型号，所以要尽量找到准确的型号名称或注明主要部件。
- 说明正在使用的相关软件。如果你在加载模块时遇到了问题，你要说明正在使用的 `kmod`、`systemd` 和 `udev` 的版本。如果其中一个 DRM 驱动出现问题，你要说明 `libdrm` 和 `Mesa` 的版本；还要说明你的 Wayland 合成器或 X-Server 及其驱动。如果你有文件系统问题，请注明相应的文件系统实用程序的版本 (`e2fsprogs`, `btrfs-progs`, `xfsprogs`……)。
- 从内核中收集可能有用的额外信息。例如，`lspci -nn` 的输出可以帮助别人识别你使用的硬件。如果你的硬件有问题，你甚至可以给出 `sudo lspci -vvv` 的结果，因为它提供了组件是如何配置的信息。对于一些问题，可能最好包含 `/proc/cpuinfo`, `/proc/ioports`, `/proc/iomem`, `/proc/modules` 或 `/proc/scsi/scsi` 等文件的内容。一些子系统还提供了收集相关信息的工具。`alsa-info.sh` 就是这样一个工具，它是音频/声音子系统开发者提供的。

这些例子应该会给你一些知识点，让你知道附上什么数据可能是明智的，但你自己也要想一想，哪些数据对别人会有帮助。不要太担心忘记一些东西，因为开发人员会要求提供他们需要的额外细节。但从一开始就把所有重要的东西都提供出来，会增加别人仔细查看的机会。

### 重要部分：报告的开头

现在你已经准备好了报告的详细部分，让我们进入最重要的部分：开头几句。现在到报告的最前面，在你刚才写的部分之前加上类似“*The detailed description:*”（详细描述）这样的内容，并在最前面插入两个新行。现在写一个正常长度的段落，大致概述这个问题。去掉所有枯燥的细节，把重点放在读者需要知道的关键部分，以让人了解这是怎么回事；如果你认为这个缺陷影响了很多用户，就提一下这点来吸引大家关注。

做好这一点后，在顶部再插入两行，写一句话的摘要，快速解释报告的内容。之后你要更加抽象，为报告写一个更短的主题/标题。

现在你已经写好了这部分，请花点时间来优化它，因为它是你的报告中最重要的部分：很多人会先读这部分，然后才会决定是否值得花时间阅读其他部分。

现在就像 MAINTAINERS 维护者文件告诉你的那样发送或提交报告，除非它是前面概述的那些“高优先级问题”之一：在这种情况下，请先阅读下一小节，然后再发送报告。

### 高优先级问题的特殊处理

高优先级问题的报告需要特殊处理。

**非常严重的缺陷：**确保在主题或工单标题以及第一段中明显标出 **severeness**（非常严重的）。

**回归：**如果问题是一个回归，请在邮件的主题或缺陷跟踪器的标题中添加 [REGRESSION]。如果您没有进行二分，请至少注明您测试的最新主线版本（比如 5.7）和出现问题的最新版本（比如 5.8）。如果您成功地进行了二分，请注明导致回归的提交 ID 和主题。也请添加该变更的作者到你的报告中；如果您需要将您的缺陷提交到缺陷跟踪器中，请将报告以私人邮件的形式转发给他，并注明报告提交地点。

**安全问题：**对于这种问题，你将必须评估：如果细节被公开披露，是否会对其他用户产生短期风险。如果不会，只需按照所述继续报告问题。如果有此风险，你需要稍微调整一下报告流程。

- 如果 MAINTAINERS 文件指示您通过邮件报告问题，请不要抄送任何公共邮件列表。
- 如果你应该在缺陷跟踪器中提交问题，请确保将工单标记为“私有”或“安全问题”。如果缺陷跟踪器没有提供保持报告私密性的方法，那就别想了，把你的报告以私人邮件的形式发送给维护者吧。

在这两种情况下，都一定要将报告发到 MAINTAINERS 文件中“安全联络”部分列出的地址。理想的情况是在发送报告的时候直接抄送他们。如果您在缺陷跟踪器中提交了报告，请将报告的文本转发到这些地址；但请在报告的顶部加上注释，表明您提交了报告，并附上工单链接。

更多信息请参见“[安全缺陷](#)”。

## 发布报告后的责任

等待别人的反应，继续推进事情，直到你能够接受这样或那样的结果。因此，请公开和及时地回应任何询问。测试提出的修复。积极地测试：至少重新测试每个新主线版本的首个候选版本（*RC*），并报告你的结果。如果出现拖延，就友好地提醒一下。如果你没有得到任何帮助或者未能满意，请试着自己帮助自己。

如果你的报告非常优秀，而且你真的很幸运，那么某个开发者可能会立即发现导致问题的原因；然后他们可能会写一个补丁来修复、测试它，并直接发送给主线集成，同时标记它以便以后回溯到需要它的稳定版和长期支持内核。那么你需要做的就是回复一句“Thank you very much”（非常感谢），然后在发布后换上修复好的版本。

但这种理想状况很少发生。这就是为什么你把报告拿出来之后工作才开始。你要做的事情要视情况而定，但通常会是下面列出的事情。但在深入研究细节之前，这里有几件重要的事情，你需要记住这部分的过程。

## 关于进一步互动的一般建议

**总是公开回复：**当你在缺陷跟踪器中提交问题时，一定要在那里回复，不要私下联系任何开发者。对于邮件报告，在回复您收到的任何邮件时，总是使用“全部回复”功能。这包括带有任何你可能想要添加到你的报告中的额外数据的邮件：进入邮件应用程序“已发送”文件夹，并在邮件上使用“全部回复”来回复报告。这种方法可以确保公共邮件列表和其他所有参与者都能及时了解情况；它还能保持邮件线程的完整性，这对于邮件列表将所有相关邮件归为一类是非常重要的。

只有两种情况不适合在缺陷跟踪器或“全部回复”中发表评论：

- 有人让你私下发东西。
- 你被告知要发送一些东西，但注意到其中包含需要保密的敏感信息。在这种情况下，可以私下发送给要求发送的开发者。但要在工单或邮件中注明你是这么做的，这样其他人就知道你尊重了这个要求。

**在请求解释或帮助之前先研究一下：**在这部分过程中，有人可能会告诉你用尚未掌握的技能做一些事情。例如你可能会被要求使用一些你从未听说过的测试工具；或者你可能会被要求在 Linux 内核源代码上应用一个补丁来测试它是否有帮助。在某些情况下，发个回复询问如何做就可以了。但在走这条路之前，尽量通过在互联网上搜索自行找到答案；或者考虑在其他地方询问建议。比如询问朋友，或者到你平时常去的聊天室或论坛发帖咨询。

**要有耐心：**如果你真的很幸运，你可能会在几个小时内收到对你的报告的答复。但大多数情况下会花费更多的时间，因为维护者分散在全球各地，因此可能在不同的时区——在那里他们已经享受着远离键盘的夜晚。

一般来说，内核开发者需要一到五个工作日来回复报告。有时会花费更长的时间，因为他们可能正忙于合并窗口、其他工作、参加开发者会议，或者只是在享受一个漫长的暑假。

“高优先级的问题”（见上面的解释）例外：维护者应该尽快解决这些问题；这就是为什么你应该最多等待一个星期（如果是紧急的事情，则只需两天），然后再发送友好的提醒。

有时维护者可能没有及时回复；有时候可能会出现分歧，例如一个问题是否符合回归的条件。在这种情况下，在邮件列表上提出你的顾虑，并请求其他人公开或私下回复如何继续推进。如果失败了，可能应该让更高级别

的维护者介入。如果是 WiFi 驱动，那就是无线维护者；如果没有更高级别的维护者，或者其他一切努力都失败了，那这可能是一种罕见的、可以让 Linus Torvalds 参与进来的情况。

**主动测试：**每当一个新的主线内核版本的第一个预发布版本（rc1）发布的时候，去检查一下这个问题是否得到了解决，或者是否有什么重要的变化。在工单中或在回复报告的邮件中提及结果（确保所有参与讨论的人都被抄送）。这将表明你的承诺和你愿意帮忙。如果问题持续存在，它也会提醒开发者确保他们不会忘记它。其他一些不定期的重新测试（例如用 rc3、rc5 和最终版本）也是一个好主意，但只有在相关的东西发生变化或者你正在写什么东西的时候才报告你的结果。

这些些常规的事情就不说了，我们来谈谈报告后如何帮助解决问题的细节。

### 查询和测试请求

如果你的报告得到了回复则需履行以下责任：

**检查与你打交道的人：**大多数情况下，会是维护者或特定代码区域的开发人员对你的报告做出回应。但由于问题是公开报告的，所以回复的可能是任何人——包括那些想要帮忙的人，但最后可能会用他们的问题或请求引导你完全偏离轨道。这很少发生，但这是快速上网搜搜看你正在与谁互动是明智之举的许多原因之一。通过这样做，你也可以知道你的报告是否被正确的人听到，因为如果讨论没有导致满意的问题解决方案而淡出，之后可能需要提醒维护者（见下文）。

**查询数据：**通常你会被要求测试一些东西或提供更多细节。尽快提供所要求的信息，因为你已经得到了可能会帮助你的人的注意，你等待的时间越长就有越可能失去关注；如果你不在数个工作日内提供信息，甚至可能出现这种结果。

**测试请求：**当你被要求测试一个诊断补丁或可能的修复时，也要尽量及时测试。但要做得恰当，一定不要急于求成：混淆事情很容易发生，这会给所有人带来许多困惑。例如一个常见的错误是以应用了一个带修复的建议补丁，但事实上并没有。即使是有经验的测试人员也会偶尔发生这样的事情，但当有修复的内核和没有修复的内核表现得一样时，他们大多时候会注意到。

### 当没有任何实质性进展时该怎么办

有些报告不会得到负有相关责任的 Linux 内核开发者的任何反应；或者围绕这个问题的讨论有所发展，但渐渐淡出，没有任何实质内容产出。

在这种情况下，要等两个星期（最好是三个星期）后再发出友好的提醒：也许当你的报告到达时，维护者刚刚离开键盘一段时间，或者有更重要的事情要处理。在写提醒信的时候，要善意地问一下，是否还需要你这边提供什么来让事情推进下去。如果报告是通过邮件发出来的，那就在邮件的第一行回复你的初始邮件（见上文），其中包括下方的原始报告的完整引用：这是少数几种情况下，这样的“TOFU”（Text Over, Fullquote Under 文字在上，完整引用在下）是正确的做法，因为这样所有的收件人都会以适当的顺序立即让细节到手头上来。

在提醒之后，再等三周的回复。如果你仍然没有得到适当的反馈，你首先应该重新考虑你的方法。你是否可能尝试接触了错误的人？是不是报告也许令人反感或者太混乱，以至于人们决定完全远离它？排除这些因素的最好方法是：把报告给一两个熟悉 FLOSS 问题报告的人看，询问他们的意见。同时征求他们关于如何继续推进的建议。这可能意味着：准备一份更好的报告，让这些人在你发出去之前对它进行审查。这样的方法完全可以；只需说明这是关于这个问题的第二份改进的报告，并附上第一份报告的链接。

如果报告是恰当的，你可以发送第二封提醒信；在其中询问为什么报告没有得到任何回复。第二封提醒邮件的好时机是在新 Linux 内核版本的首个预发布版本（‘rc1’）发布后不久，因为无论如何你都应该在那个时候重新测试并提供状态更新（见上文）。

如果第二次提醒的结果又在一周内没有任何反应，可以尝试联系上级维护者询问意见：即使再忙的维护者在这时候也至少应该发过某种确认。

记住要做好失望的准备：理想状况下维护者最好对每一个问题报告做出回应，但他们只有义务解决之前列出的“高优先级问题”。所以，如果你得到的回复是“谢谢你的报告，我目前有更重要的问题要处理，在可预见的未来没有时间去研究这个问题”，那请不要太沮丧。

也有可能在缺陷跟踪器或列表中进行了一些讨论之后，什么都没有发生，提醒也无助于激励大家进行修复。这种情况可能是毁灭性的，但在 Linux 内核开发中确实会发生。这些和其他得不到帮助的原因在本文结尾处的“为什么有些问题在被报告后没有得到任何回应或者仍然没有修复”中进行了解释。

如果你没有得到任何帮助或问题最终没有得到解决，不要沮丧：Linux 内核是 FLOSS，因此你仍然可以自己帮助自己。例如，你可以试着找到其他受影响的人，和他们一起合作来解决这个问题。这样的团队可以一起准备一份新的报告，提到团队有多少人，为什么你们认为这是应该得到解决的事情。也许你们还可以一起缩小确切原因或引入回归的变化，这往往会使修复更容易。而且如果运气好的话，团队中可能会有懂点编程的人，也许能写出一个修复方案。

## “报告稳定版和长期支持内核线的回归”的参考

本小节提供了在稳定版和长期支持内核线中面对回归时需要执行的步骤的详细信息。

### 确保特定版本线仍然受支持

检查内核开发人员是否仍然维护你关心的 Linux 内核版本线：去 [kernel.org](http://kernel.org) 的首页，确保此特定版本线的最新版没有 “[EOL]” 标记。

大多数内核版本线只支持三个月左右，因为延长维护时间会带来相当多的工作。因此，每年只会选择一个版本来支持至少两年（通常是六年）。这就是为什么你需要检查内核开发者是否还支持你关心的版本线。

注意，如果 [kernel.org](http://kernel.org) 在首页上列出了两个“稳定”版本，你应该考虑切换到较新的版本，而忘掉较旧的版本：对它的支持可能很快就会结束。然后，它将被标记为“生命周期结束”（EOL）。达到这个程度的版本线仍然会在 [kernel.org](http://kernel.org) 首页上被显示一两周，但不适合用于测试和报告。

### 搜索稳定版邮件列表

检查 Linux 稳定版邮件列表中的现有报告。

也许你所面临的问题已经被发现，并且已经或即将被修复。因此，请在 Linux 稳定版邮件列表的档案中搜索类似问题的报告。如果你找到任何匹配的问题，可以考虑加入讨论，除非修复工作已经完成并计划很快得到应用。

### 用最新版本复现问题

从特定的版本线安装最新版本作为纯净内核。确保这个内核没有被污染，并且仍然存在问题，因为问题可能已经在那被修复了。

在投入更多时间到这个过程中之前，你要检查这个问题是否在你关注的版本线的最新版本中已经得到了修复。这个内核需要是纯净的，在问题发生之前不应该被污染，正如上面已经在测试主线的过程中详细介绍过的一样。

您是否是第一次注意到供应商内核的回归？供应商的更改可能会发生变化。你需要重新检查排除来这个问题。当您从 5.10.4-vendor.42 更新到 5.10.5-vendor.43 时，记录损坏的信息。然后在测试了前一段中所述的最新 5.10 版本之后，检查 Linux 5.10.4 的普通版本是否也可以正常工作。如果问题在那里出现，那就不符合上游回归的条件，您需要切换回主逐步指南来报告问题。

### 报告回归

向 Linux 稳定版邮件列表发送一个简短的问题报告 ([stable@vger.kernel.org](mailto:stable@vger.kernel.org))。大致描述问题，并解释如何复现。讲清楚首个出现问题的版本和最后一个工作正常的版本。然后等待进一步的指示。

当报告在稳定版或长期支持内核线内发生的回归（例如在从 5.10.4 更新到 5.10.5 时），一份简短的报告足以快速报告问题。因此只需要粗略的描述。

但是请注意，如果您能够指明引入问题的确切版本，这将对开发人员有很大帮助。因此如果有时间的话，请尝试使用普通内核找到该版本。让我们假设发行版发布 Linux 内核 5.10.5 到 5.10.8 的更新时发生了故障。那么按照上面的指示，去检查该版本线中的最新内核，比如 5.10.9。如果问题出现，请尝试普通 5.10.5，以确保供应商应用的补丁不会干扰。如果问题没有出现，那么尝试 5.10.7，然后直到 5.10.8 或 5.10.6（取决于结果）找到第一个引入问题的版本。在报告中写明这一点，并指出 5.10.9 仍然存在故障。

前一段基本粗略地概述了“二分”方法。一旦报告出来，您可能被要求做一个正确的报告，因为它允许精确地定位导致问题的确切更改（然后很容易被恢复以快速修复问题）。因此如果时间允许，考虑立即进行适当的二分。有关如何详细信息，请参阅“对回归的特别关照”部分和文档“[二分 \(bisect\) 缺陷](#)”。

### “报告仅在旧内核版本线中发生的问题”的参考

本节详细介绍了如果无法用主线内核重现问题，但希望在旧版本线（又称稳定版内核和长期支持内核）中修复问题时需要采取的步骤。

### 有些修复太复杂

请做好准备，接下来的几个步骤可能无法在旧版本中解决问题：修复可能太大或太冒险，无法移植到那里。

即使是微小的、看似明显的代码变化，有时也会带来新的、完全意想不到的问题。稳定版和长期支持内核的维护者非常清楚这一点，因此他们只对这些内核进行符合“[所有你想知道的事情 - 关于 linux 稳定版发布](#)”中所列出的规则的修改。

复杂或有风险的修改不符合条件，因此只能应用于主线。其他的修复很容易被回溯到最新的稳定版和长期支持内核，但是风险太大，无法集成到旧版内核中。所以要注意你所希望的修复可能是那些不会被回溯到你所关心的版本线的修复之一。在这种情况下，你将别无选择，要么忍受这个问题，要么切换到一个较新的 Linux 版本，除非你想自己把修复补丁应用到你的内核中。

## 通用准备

执行上面“报告仅在旧内核版本线中发生的问题”一节中的前三个步骤。

您需要执行本指南另一节中已经描述的几个步骤。这些步骤将让您：

- 检查内核开发人员是否仍然维护您关心的 Linux 内核版本行。
- 在 Linux 稳定邮件列表中搜索退出的报告。
- 检查最新版本。

## 检查代码历史和搜索现有的讨论

在 Linux 内核版本控制系统中搜索修复主线问题的更改，因为它的提交消息可能会告诉你修复是否已经计划好了支持。如果你没有找到，搜索适当的邮件列表，寻找讨论此类问题或同行评议可能修复的帖子；然后检查讨论是否认为修复不适合支持。如果支持根本不被考虑，加入最新的讨论，询问是否有可能。

在许多情况下，你所处理的问题会发生在主线上，但已在主线上得到了解决。修正它的提交也需要被回溯才能解决这个问题。这就是为什么你要搜索它或任何相关讨论。

- 首先尝试在存放 Linux 内核源代码的 Git 仓库中找到修复。你可以通过 [kernel.org 上的网页](#) 或 [GitHub 上的镜像](#) 来实现；如果你有一个本地克隆，你也可以在命令行用 `git log --grep=<pattern>` 来搜索。

如果你找到了修复，请查看提交消息的尾部是否包含了类似这样的“稳定版标签”：

Cc: <[sstable@vger.kernel.org](mailto:sstable@vger.kernel.org)> # 5.4+

像上面这行，开发者标记了安全修复可以回传到 5.4 及以后的版本。大多数情况下，它会在两周内被应用到那里，但有时需要更长的时间。

- 如果提交没有告诉你任何东西，或者你找不到修复，请再找找关于这个问题的讨论。用你最喜欢的搜索引擎搜索网络，以及 [Linux kernel developers mailing list](#) 内核开发者邮件列表 的档案。也可以阅读上面的 定位导致问题的内核区域一节，然后按照说明找到导致问题的子系统：它的缺陷跟踪器或邮件列表存档中可能有你要找的答案。
- 如果你看到了一个计划的修复，请按上所述在版本控制系统中搜索它，因为提交可能会告诉你是否可以进行回溯。
  - 检查讨论中是否有任何迹象表明，该修复程序可能风险太大，无法回溯到你关心的版本线。如果是这样的话，你必须忍受这个问题，或者切换到应用了修复的内核版本线。

- 如果修复的问题未包含稳定版标签，并且没有讨论过回溯问题，请加入讨论：如果合适的话，请提及你所面对的问题的版本，以及你希望看到它被修复。

## 请求建议

前面的步骤之一应该会给出一个解决方案。如果仍未能成功，请向可能引起问题的子系统的维护人员询问建议；抄送特定子系统的邮件列表以及稳定版邮件列表。

如果前面的三个步骤都没有让你更接近解决方案，那么只剩下一个选择：请求建议。在你发给可能是问题根源的子系统的维护者的邮件中这样做；抄送子系统的邮件列表以及稳定版邮件列表 ([stable@vger.kernel.org](mailto:stable@vger.kernel.org))。

## 为什么有些问题在报告后没有任何回应或仍未解决？

当向 Linux 开发者报告问题时，要注意只有“高优先级的问题”（回归、安全问题、严重问题）才一定会得到解决。如果维护者或其他人都失败了，Linus Torvalds 他自己会确保这一点。他们和其他内核开发者也会解决很多其他问题。但是要知道，有时他们也会不能或不愿帮忙；有时甚至没有人发报告给他们。

最好的解释就是那些内核开发者常常是在业余时间为 Linux 内核做出贡献。内核中的不少驱动程序都是由这样的程序员编写的，往往只是因为他们想让自己的硬件可以在自己喜欢的操作系统上使用。

这些程序员大多数时候会很乐意修复别人报告的问题。但是没有人可以强迫他们这样做，因为他们是自愿贡献的。

还有一些情况下，这些开发者真的很想解决一个问题，但却不能解决：有时他们缺乏硬件编程文档来解决问题。这种情况往往由于公开的文档太简陋，或者驱动程序是通过逆向工程编写的。

业余开发者迟早也会不再关心某驱动。也许他们的测试硬件坏了，被更高级的玩意取代了，或者是太老了以至于只能在计算机博物馆里找到。有时开发者根本就不关心他们的代码和 Linux 了，因为在他们的生活中一些不同的东西变得更重要了。在某些情况下，没有人愿意接手维护者的工作——也没有人可以被强迫，因为对 Linux 内核的贡献是自愿的。然而被遗弃的驱动程序仍然存在于内核中：它们对人们仍然有用，删除它们可能导致回归。

对于那些为 Linux 内核工作而获得报酬的开发者来说，情况并没有什么不同。这些人现在贡献了大部分的变更。但是他们的雇主迟早也会停止关注他们的代码或者让程序员专注于其他事情。例如，硬件厂商主要通过销售新硬件来赚钱；因此，他们中的不少人并没有投入太多时间和精力来维护他们多年前就停止销售的东西的 Linux 内核驱动。企业级 Linux 发行商往往持续维护的时间比较长，但在新版本中往往把对老旧和稀有硬件的支持放在一边，以限制范围。一旦公司抛弃了一些代码，往往由业余贡献者接手，但正如上面提到的：他们迟早也会放下代码。

优先级是一些问题没有被修复的另一个原因，因为维护者相当多的时候是被迫设置这些优先级的，因为在 Linux 上工作的时间是有限的。对于业余时间或者雇主给予他们的开发人员用于上游内核维护工作的时间也是如此。有时维护人员也会被报告淹没，即使一个驱动程序几乎完美地工作。为了不被完全缠住，程序员可能别无选择，只能对问题报告进行优先级排序而拒绝其中的一些报告。

不过这些都不用太过担心，很多驱动都有积极的维护者，他们对尽可能多的解决问题相当感兴趣。

## 结束语

与其他免费/自由 & 开源软件 (Free/Libre & Open Source Software, FLOSS) 相比, 向 Linux 内核开发者报告问题是很难的: 这个文档的长度和复杂性以及字里行间的内涵都说明了这一点。但目前就是这样了。这篇文字的主要作者希望通过记录现状来为以后改善这种状况打下一些基础。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解, 而不是作为一个分支。因此, 如果您对此文件有任何意见或更新, 请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题, 请联系该文件的译者, 或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** [.../..../admin-guide/security-bugs](#)

译者 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

## 安全缺陷

Linux 内核开发人员非常重视安全性。因此我们想知道何时发现了安全漏洞, 以便尽快修复和披露。请向 Linux 内核安全团队报告安全漏洞。

## 联络

可以通过电子邮件 <[security@kernel.org](mailto:security@kernel.org)> 联系 Linux 内核安全团队。这是一个安全人员的私有列表, 他们将帮助验证错误报告并开发和发布修复程序。如果您已经有了一个修复, 请将其包含在您的报告中, 这样可以大大加快进程。安全团队可能会从区域维护人员那里获得额外的帮助, 以理解和修复安全漏洞。

与任何缺陷一样, 提供的信息越多, 诊断和修复就越容易。如果您不清楚哪些信息有用, 请查看 “[报告问题](#)” 中概述的步骤。任何利用漏洞的攻击代码都非常有用, 未经报告者同意不会对外发布, 除非已经公开。

请尽可能发送无附件的纯文本电子邮件。如果所有的细节都藏在附件里, 那么就很难对一个复杂的问题进行上下文引用的讨论。把它想象成一个[常规的补丁提交](#) (即使你还没有补丁): 描述问题和影响, 列出复现步骤, 然后给出一个建议的解决方案, 所有这些都是纯文本的。

### 披露和限制信息

安全列表不是公开渠道。为此，请参见下面的协作。

一旦开发出了健壮的补丁，发布过程就开始了。对公开的缺陷的修复会立即发布。

尽管我们倾向于在未公开缺陷的修复可用时即发布补丁，但应报告者或受影响方的请求，这可能会被推迟到发布过程开始后的 7 日内，如果根据缺陷的严重性需要更多的时间，则可额外延长到 14 天。推迟发布修复的唯一有效原因是为了解决 QA 的逻辑和需要发布协调的大规模部署。

虽然可能与受信任的个人共享受限信息以开发修复，但未经报告者许可，此类信息不会与修复程序一起发布或发布在任何其他披露渠道上。这包括但不限于原始错误报告和后续讨论（如有）、漏洞、CVE 信息或报告者的身份。

换句话说，我们唯一感兴趣的是修复缺陷。提交给安全列表的所有其他资料以及对报告的任何后续讨论，即使在解除限制之后，也将永久保密。

### 协调

对敏感缺陷（例如那些可能导致权限提升的缺陷）的修复可能需要与私有邮件列表 <[linux-distros@vs.openwall.org](mailto:linux-distros@vs.openwall.org)> 进行协调，以便分发供应商做好准备，在公开披露上游补丁时发布一个已修复的内核。发行版将需要一些时间来测试建议的补丁，通常会要求至少几天的限制，而供应商更新发布更倾向于周二至周四。若合适，安全团队可以协助这种协调，或者报告者可以从一开始就包括 linux 发行版。在这种情况下，请记住在电子邮件主题行前面加上 “[vs]” ，如 linux 发行版 wiki 中所述：[<http://oss-security.openwall.org/wiki/mailing-lists/distros#how-to-use-the-lists>](http://oss-security.openwall.org/wiki/mailing-lists/distros#how-to-use-the-lists)。

### CVE 分配

安全团队通常不分配 CVE，我们也不需要它们来进行报告或修复，因为这会使过程不必要的复杂化，并可能耽误缺陷处理。如果报告者希望在公开披露之前分配一个 CVE 编号，他们需要联系上述的私有 linux-distros 列表。当在提供补丁之前已有这样的 CVE 编号时，如报告者愿意，最好在提交消息中提及它。

### 保密协议

Linux 内核安全团队不是一个正式的机构实体，因此无法签订任何保密协议。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的

帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** .../.../admin-guide/bug-hunting

译者 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

## 追踪缺陷

内核错误报告通常附带如下堆栈转储:

```
-----[ cut here ]-----
WARNING: CPU: 1 PID: 28102 at kernel/module.c:1108 module_put+0x57/0x70
Modules linked in: dvb_usb_gp8psk(-) dvb_usb dvb_core nvidia_drm(P0) nvidia_modeset(P0) snd_
↳ hda_codec_hdmi snd_hda_intel snd_hda_codec snd_hwdep snd_hda_core snd_pcm snd_timer snd_
↳ soundcore nvidia(P0) [last unloaded: rc_core]
CPU: 1 PID: 28102 Comm: rmmmod Tainted: P WC 0 4.8.4-build.1 #1
Hardware name: MSI MS-7309/MS-7309, BIOS V1.12 02/23/2009
00000000 c12ba080 00000000 00000000 c103ed6a c1616014 00000001 00006dc6
c1615862 00000454 c109e8a7 c109e8a7 00000009 ffffffff 00000000 f13f6a10
f5f5a600 c103ee33 00000009 00000000 00000000 c109e8a7 f80ca4d0 c109f617
Call Trace:
[<c12ba080>] ? dump_stack+0x44/0x64
[<c103ed6a>] ? __warn+0xfa/0x120
[<c109e8a7>] ? module_put+0x57/0x70
[<c109e8a7>] ? module_put+0x57/0x70
[<c103ee33>] ? warn_slowpath_null+0x23/0x30
[<c109e8a7>] ? module_put+0x57/0x70
[<f80ca4d0>] ? gp8psk_fe_set_frontend+0x460/0x460 [dvb_usb_gp8psk]
[<c109f617>] ? symbol_put_addr+0x27/0x50
[<f80bc9ca>] ? dvb_usb_adapter_frontend_exit+0x3a/0x70 [dvb_usb]
[<f80bb3bf>] ? dvb_usb_exit+0x2f/0xd0 [dvb_usb]
[<c13d03bc>] ? usb_disable_endpoint+0x7c/0xb0
[<f80bb48a>] ? dvb_usb_device_exit+0x2a/0x50 [dvb_usb]
[<c13d2882>] ? usb_unbind_interface+0x62/0x250
[<c136b514>] ? __pm_runtime_idle+0x44/0x70
[<c13620d8>] ? __device_release_driver+0x78/0x120
[<c1362907>] ? driver_detach+0x87/0x90
[<c1361c48>] ? bus_remove_driver+0x38/0x90
[<c13d1c18>] ? usb_deregister+0x58/0xb0
[<c109fb0>] ? SyS_delete_module+0x130/0x1f0
[<c1055654>] ? task_work_run+0x64/0x80
[<c1000fa5>] ? exit_to_usermode_loop+0x85/0x90
[<c10013f0>] ? do_fast_syscall_32+0x80/0x130
[<c1549f43>] ? sysenter_past_esp+0x40/0x6a
---[ end trace 6ebc60ef3981792f ]---
```

这样的堆栈跟踪提供了足够的信息来识别内核源代码中发生错误的那一行。根据问题的严重性，它还可能包含“**Oops**”一词，比如:

```
BUG: unable to handle kernel NULL pointer dereference at  (null)
IP: [<c06969d4>] iret_exc+0x7d0/0xa59
*pdp = 000000002258a001 *pde = 0000000000000000
Oops: 0002 [#1] PREEMPT SMP
...
...
```

尽管有 **Oops** 或其他类型的堆栈跟踪，但通常需要找到出问题的行来识别和处理缺陷。在本章中，我们将参考“Oops”来了解需要分析的各种堆栈跟踪。

如果内核是用 `CONFIG_DEBUG_INFO` 编译的，那么可以使用文件：`scripts/decode_stacktrace.sh`。

### 链接的模块

受到污染或正在加载/卸载的模块用“(...)" 标记，污染标志在 `Documentation/admin-guide/tainted-kernels.rst` 文件中进行了描述，“正在被加载”用“+”标注，“正在被卸载”用“-”标注。

### Oops 消息在哪？

通常，Oops 文本由 `klogd` 从内核缓冲区读取，然后交给 `syslogd`，后者将其写入 `syslog` 文件，通常是 `/var/log/messages`（取决于 `/etc/syslog.conf`）。在使用 `systemd` 的系统上，它也可以由 `journald` 守护进程存储，并通过运行 `journalctl` 命令进行访问。

有时 `klogd` 会挂掉，这种情况下您可以运行 `dmesg > file` 从内核缓冲区读取数据并保存它。或者您可以 `cat /proc/kmsg > file`，但是您必须适时中断以停止传输，因为 `kmsg` 是一个“永无止境的文件”。

如果机器严重崩溃，无法输入命令或磁盘不可用，那还有三个选项：

- (1) 手动复制屏幕上的文本，并在机器重新启动后输入。很难受，但这是突然崩溃下唯一的选择。或者你可以用数码相机拍下屏幕——虽然不那么好，但总比什么都没有好。如果消息滚动超出控制台顶部，使用更高分辨率（例如 `vga=791`）引导启动将允许您阅读更多文本。（警告：这需要 `vesafb`，因此对“早期”的 Oppses 没有帮助）
- (2) 从串口终端启动（参见 `Documentation/admin-guide/serial-console.rst`），在另一台机器上运行调制解调器然后用你喜欢的通信程序捕获输出。`Minicom` 运行良好。
- (3) 使用 `Kdump`（参阅 `Documentation/admin-guide/kdump/kdump.rst`），使用 `Documentation/admin-guide/kdump/gdbmacros.txt` 中的 `dmesg gdbmacro` 从旧内存中提取内核环形缓冲区。

## 找到缺陷位置

如果你能指出缺陷在内核源代码中的位置，则报告缺陷的效果会非常好。这有两种方法。通常来说使用 `gdb` 会比较容易，不过内核需要用调试信息来预编译。

### gdb

GNU 调试器 (GNU debugger, `gdb`) 是从 `vmlinux` 文件中找出 OOPS 的确切文件和行号的最佳方法。

在使用 `CONFIG_DEBUG_INFO` 编译的内核上使用 `gdb` 效果最好。可通过运行以下命令进行设置：

```
$ ./scripts/config -d COMPILE_TEST -e DEBUG_KERNEL -e DEBUG_INFO
```

在用 `CONFIG_DEBUG_INFO` 编译的内核上，你可以直接从 OOPS 复制 EIP 值：

```
EIP: 0060:[<c021e50e>] Not tainted VLI
```

并使用 GDB 来将其翻译成可读形式：

```
$ gdb vmlinux
(gdb) l *0xc021e50e
```

如果没有启用 `CONFIG_DEBUG_INFO`，则使用 OOPS 的函数偏移：

```
EIP is at vt_ioctl+0xda8/0x1482
```

并在启用 `CONFIG_DEBUG_INFO` 的情况下重新编译内核：

```
$ ./scripts/config -d COMPILE_TEST -e DEBUG_KERNEL -e DEBUG_INFO
$ make vmlinux
$ gdb vmlinux
(gdb) l *vt_ioctl+0xda8
0x1888 is in vt_ioctl (drivers/tty/vt/vt_ioctl.c:293).
288 {
289     struct vc_data *vc = NULL;
290     int ret = 0;
291
292     console_lock();
293     if (VT_BUSY(vc_num))
294         ret = -EBUSY;
295     else if (vc_num)
296         vc = vc_deallocate(vc_num);
297     console_unlock();
```

或者若您想要更详细的显示：

```
(gdb) p vt_ioctl
$1 = {int (struct tty_struct *, unsigned int, unsigned long)} 0xae0 <vt_ioctl>
(gdb) l *0xae0+0xda8
```

您也可以使用对象文件作为替代:

```
$ make drivers/tty/  
$ gdb drivers/tty/vt/vt_ioctl.o  
(gdb) l *vt_ioctl+0xda8
```

如果你有调用跟踪，类似:

```
Call Trace:  
[<fffffffff8802c8e9>] :jbd:log_wait_commit+0xa3/0xf5  
[<fffffffff810482d9>] autoremove_wake_function+0x0/0x2e  
[<fffffffff8802770b>] :jbd:journal_stop+0x1be/0x1ee  
...
```

这表明问题可能在:jbd: 模块中。您可以在 gdb 中加载该模块并列出相关代码:

```
$ gdb fs/jbd/jbd.ko  
(gdb) l *log_wait_commit+0xa3
```

---

**Note:** 您还可以对堆栈跟踪处的任何函数调用执行相同的操作，例如:

```
[<f80bc9ca>] ? dvb_usb_adapter_frontend_exit+0x3a/0x70 [dvb_usb]
```

上述调用发生的位置可以通过以下方式看到:

```
$ gdb drivers/media/usb/dvb-usb/dvb-usb.o  
(gdb) l *dvb_usb_adapter_frontend_exit+0x3a
```

---

## objdump

要调试内核，请使用 objdump 并从崩溃输出中查找十六进制偏移，以找到有效的代码/汇编行。如果没有调试符号，您将看到所示例程的汇编程序代码，但是如果内核有调试符号，C 代码也将可见（调试符号可以在内核配置菜单的 hacking 项中启用）。例如:

```
$ objdump -r -S -l --disassemble net/dccp/ipv4.o
```

---

**Note:** 您需要处于内核树的顶层以便此获得您的 C 文件。

---

如果您无法访问源代码，仍然可以使用以下方法调试一些崩溃转储（如 Dave Miller 的示例崩溃转储输出所示）：

```
EIP is at +0x14/0x4c0
```

```
...
Code: 44 24 04 e8 6f 05 00 00 e9 e8 fe ff ff 8d 76 00 8d bc 27 00 00
00 00 55 57 56 53 81 ec bc 00 00 00 8b ac 24 d0 00 00 00 8b 5d 08
<8b> 83 3c 01 00 00 89 44 24 14 8b 45 28 85 c0 89 44 24 18 0f 85
```

Put the bytes into a "foo.s" file like this:

```
.text
.globl foo
foo:
    .byte .... /* bytes from Code: part of OOPS dump */
```

Compile it with "gcc -c -o foo.o foo.s" then look at the output of "objdump --disassemble foo.o".

Output:

```
ip_queue_xmit:
push    %ebp
push    %edi
push    %esi
push    %ebx
sub    $0xbc, %esp
mov    0xd0(%esp), %ebp      ! %ebp = arg0 (skb)
mov    0x8(%ebp), %ebx      ! %ebx = skb->sk
mov    0x13c(%ebx), %eax    ! %eax = inet_sk(sk)->opt
```

*scripts/decode decode* 文件可以用来自动完成大部分工作，这取决于正在调试的 CPU 体系结构。

## 报告缺陷

一旦你通过定位缺陷找到了其发生的地方，你可以尝试自己修复它或者向上游报告它。

为了向上游报告，您应该找出用于开发受影响代码的邮件列表。这可以使用 `get_maintainer.pl`。

例如，您在 gspca 的 sonixj.c 文件中发现一个缺陷，则可以通过以下方法找到它的维护者：

```
$ ./scripts/get_maintainer.pl -f drivers/media/usb/gspca/sonixj.c
Hans Verkuil <hverkuil@xs4all.nl> (odd fixer:GSPCA USB WEBCAM DRIVER,commit_signer:1/1=100%)
Mauro Carvalho Chehab <mcchehab@kernel.org> (maintainer:MEDIA INPUT INFRASTRUCTURE (V4L/DVB),
➥commit_signer:1/1=100%)
Tejun Heo <tj@kernel.org> (commit_signer:1/1=100%)
Bhaktipriya Shridhar <bhaktipriya96@gmail.com> (commit_signer:1/1=100%,authored:1/1=100%,added_
➥lines:4/4=100%,removed_lines:9/9=100%)
linux-media@vger.kernel.org (open list:GSPCA USB WEBCAM DRIVER)
linux-kernel@vger.kernel.org (open list)
```

请注意它将指出：

- 最后接触源代码的开发人员（如果这是在 git 树中完成的）。在上面的例子中是 Tejun 和 Bhaktipriya（在这个特定的案例中，没有人真正参与这个文件的开发）；
- 驱动维护人员（Hans Verkuil）；
- 子系统维护人员（Mauro Carvalho Chehab）；
- 驱动程序和/或子系统邮件列表（[linux-media@vger.kernel.org](mailto:linux-media@vger.kernel.org)）；
- Linux 内核邮件列表（[linux-kernel@vger.kernel.org](mailto:linux-kernel@vger.kernel.org)）。

通常，修复缺陷的最快方法是将它报告给用于开发相关代码的邮件列表（linux-media ML），抄送驱动程序维护者（Hans）。

如果你完全不知道该把报告寄给谁，且 `get_maintainer.pl` 也没有提供任何有用的信息，请发送到 [linux-kernel@vger.kernel.org](mailto:linux-kernel@vger.kernel.org)。

感谢您的帮助，这使 Linux 尽可能稳定:-)

## 修复缺陷

如果你懂得编程，你不仅可以通过报告错误来帮助我们，还可以提供一个解决方案。毕竟，开源就是分享你的工作，你不想因为你的天才而被认可吗？

如果你决定这样做，请在制定解决方案后将其提交到上游。

请务必阅读 `Documentation/process/submitting-patches.rst`，以帮助您的代码被接受。

---

## 用 `klogd` 进行 Oops 跟踪的注意事项

为了帮助 Linus 和其他内核开发人员，`klogd` 对保护故障的处理提供了大量支持。为了完整支持地址解析，至少应该使用 `sysklogd` 包的 1.3-pl3 版本。

当发生保护故障时，`klogd` 守护进程会自动将内核日志消息中的重要地址转换为它们的等效符号。然后通过 `klogd` 使用的任何报告机制来转发这个已翻译的内核消息。保护错误消息可以直接从消息文件中剪切出来并转发给内核开发人员。

`klogd` 执行两种类型的地址解析，静态翻译和动态翻译。静态翻译使用 `System.map` 文件。为了进行静态转换，`klogd` 守护进程必须能够在守护进程初始化时找到系统映射文件。有关 `klogd` 如何搜索映射文件的信息，请参见 `klogd` 手册页。

当使用内核可加载模块时，动态地址转换非常重要。由于内核模块的内存是从内核的动态内存池中分配的，因此无论是模块的开头还是模块中的函数和符号都没有固定的位置。

内核支持系统调用，允许程序确定加载哪些模块及其在内存中的位置。`klogd` 守护进程使用这些系统调用构建了一个符号表，可用于调试可加载内核模块中发生的保护错误。

`klogd` 至少会提供产生保护故障的模块的名称。如果可加载模块的开发人员选择从模块导出符号信息，则可能会有其他可用的符号信息。

由于内核模块环境可以是动态的，因此当模块环境发生变化时，必须有一种通知 `klogd` 守护进程的机制。有一些可用的命令行选项允许 `klogd` 向当前正在执行的守护进程发出信号示意应该刷新符号信息。有关更多信息，请参阅 `klogd` 手册页。

`sysklogd` 发行版附带了一个补丁，它修改了 `modules-2.0.0` 包，以便在加载或卸载模块时自动向 `klogd` 发送信号。应用此补丁基本上可无缝支持调试内核可加载模块发生的保护故障。

以下是 `klogd` 处理的可加载模块中的保护故障示例：

```
Aug 29 09:51:01 blizard kernel: Unable to handle kernel paging request at virtual address ↵
↳ f15e97cc
Aug 29 09:51:01 blizard kernel: current->tss.cr3 = 0062d000, %cr3 = 0062d000
Aug 29 09:51:01 blizard kernel: *pde = 00000000
Aug 29 09:51:01 blizard kernel: Oops: 0002
Aug 29 09:51:01 blizard kernel: CPU: 0
Aug 29 09:51:01 blizard kernel: EIP: 0010:[oops:_oops+16/3868]
Aug 29 09:51:01 blizard kernel: EFLAGS: 00010212
Aug 29 09:51:01 blizard kernel: eax: 315e97cc ebx: 003a6f80 ecx: 001be77b edx: 00237c0c
Aug 29 09:51:01 blizard kernel: esi: 00000000 edi: bfffffdb3 ebp: 00589f90 esp: 00589f8c
Aug 29 09:51:01 blizard kernel: ds: 0018 es: 0018 fs: 002b gs: 002b ss: 0018
Aug 29 09:51:01 blizard kernel: Process oops_test (pid: 3374, process nr: 21, ↳
↳ stackpage=00589000)
Aug 29 09:51:01 blizard kernel: Stack: 315e97cc 00589f98 0100b0b4 bfffffed4 0012e38e 00240c64 ↳
↳ 003a6f80 00000001
Aug 29 09:51:01 blizard kernel: 00000000 00237810 bfffff00 0010a7fa 00000003 00000001 ↳
↳ 00000000 bfffff00
Aug 29 09:51:01 blizard kernel: bfffffdb3 bfffffed4 ffffffd4 0000002b 0007002b 0000002b ↳
↳ 0000002b 00000036
Aug 29 09:51:01 blizard kernel: Call Trace: [oops:_oops_ioctl+48/80] [_sys_ioctl+254/272] [_
↳ system_call+82/128]
Aug 29 09:51:01 blizard kernel: Code: c7 00 05 00 00 00 eb 08 90 90 90 90 90 90 89 ec 5d ↳
↳ c3
```

Dr. G.W. Wettstein  
Roger Maris Cancer Center  
820 4th St. N.  
Fargo, ND 58122  
Phone: 701-234-7556

Oncology Research Div. Computing Facility  
INTERNET: greg@wind.rmcc.com

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** ../../admin-guide/bug-bisect

译者 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

## 二分 (bisect) 缺陷

(英文版) 最后更新: 2016 年 10 月 28 日

### 引言

始终尝试由来自 kernel.org 的源代码构建的最新内核。如果您没有信心这样做, 请将错误报告给您的发行版供应商, 而不是内核开发人员。

找到缺陷 (bug) 并不总是那么容易, 不过仍然得去找。如果你找不到它, 不要放弃。尽可能多的向相关维护人员报告您发现的信息。请参阅 MAINTAINERS 文件以了解您所关注的子系统的维护人员。

在提交错误报告之前, 请阅读 “Documentation/admin-guide/reporting-issues.rst”。

### 设备未出现 (Devices not appearing)

这通常是由 udev/systemd 引起的。在将其归咎于内核之前先检查一下。

### 查找导致缺陷的补丁

使用 git 提供的工具可以很容易地找到缺陷, 只要缺陷是可复现的。

操作步骤:

- 从 git 源代码构建内核
- 以此开始二分<sup>1</sup>:

```
$ git bisect start
```

- 标记损坏的变更集:

```
$ git bisect bad [commit]
```

- 标记正常工作的变更集:

```
$ git bisect good [commit]
```

- 重新构建内核并测试
- 使用以下任一与 git bisect 进行交互:

<sup>1</sup> 您可以 (可选地) 在开始 git bisect 的时候提供 good 或 bad 参数 git bisect start [BAD] [GOOD]

```
$ git bisect good
```

或:

```
$ git bisect bad
```

这取决于您测试的变更集上是否有缺陷

- 在一些交互之后, git bisect 将给出可能导致缺陷的变更集。
- 例如, 如果您知道当前版本有问题, 而 4.8 版本是正常的, 则可以执行以下操作:

```
$ git bisect start
$ git bisect bad          # Current version is bad
$ git bisect good v4.8
```

如需进一步参考, 请阅读:

- [git-bisect 的手册页](#)
- [Fighting regressions with git bisect \(用 git bisect 解决回归\)](#)
- [Fully automated bisecting with “git bisect run” \(使用 git bisect run 来全自动二分\)](#)
- [Using Git bisect to figure out when brokenness was introduced \(使用 Git 二分来找出何时引入了错误\)](#)

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解, 而不是作为一个分支。因此, 如果您对此文件有任何意见或更新, 请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题, 请联系该文件的译者, 或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** [.../..../admin-guide/tainted-kernels](#)

译者 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

## 受污染的内核

当发生一些在稍后调查问题时可能相关的事件时, 内核会将自己标记为“受污染 (tainted)”的。不用太过担心, 大多数情况下运行受污染的内核没有问题; 这些信息主要在有人想调查某个问题时才有意义的, 因为问题的真正原因可能是导致内核受污染的事件。这就是为什么来自受污染内核的缺陷报告常常被开发人员忽略, 因此请尝试用未受污染的内核重现问题。

请注意, 即使在您消除导致污染的原因 (亦即卸载专有内核模块) 之后, 内核仍将保持污染状态, 以表示内核仍然不可信。这也是为什么内核在注意到内部问题 (“kernel bug”)、可恢复错误 (“kernel oops”) 或不可

恢复错误（“kernel panic”）时会打印受污染状态，并将有关此的调试信息写入日志 `dmesg` 输出。也可以通过 `/proc/` 中的文件在运行时检查受污染的状态。

### BUG、Oops 或 Panics 消息中的污染标志

在顶部以“CPU:”开头的一行中可以找到受污染的状态；内核是否受到污染和原因会显示在进程 ID（“PID:”）和触发事件命令的缩写名称（“Comm:”）之后：

```
BUG: unable to handle kernel NULL pointer dereference at 0000000000000000
Oops: 0002 [#1] SMP PTI
CPU: 0 PID: 4424 Comm: insmod Tainted: P      W 0      4.20.0-0.rc6.fc30 #1
Hardware name: Red Hat KVM, BIOS 0.5.1 01/01/2011
RIP: 0010:my_oops_init+0x13/0x1000 [kpanic]
[...]
```

如果内核在事件发生时没有被污染，您将在那里看到“Not-tainted:”；如果被污染，那么它将是“Tainted:”以及字母或空格。在上面的例子中，它看起来是这样的：

```
Tainted: P      W 0
```

下表解释了这些字符的含义。在本例中，由于加载了专有模块（P），出现了警告（W），并且加载了外部构建的模块（0），所以内核早些时候受到了污染。要解码其他字符，请使用下表。

### 解码运行时的污染状态

在运行时，您可以通过读取 `cat /proc/sys/kernel/tainted` 来查询受污染状态。如果返回 0，则内核没有受到污染；任何其他数字都表示受到污染的原因。解码这个数字的最简单方法是使用脚本 `tools/debugging/kernel-chktaint`，您的发行版可能会将其作为名为 `linux-tools` 或 `kernel-tools` 的包的一部分提供；如果没有，您可以从 [git.kernel.org](https://git.kernel.org) 网站下载此脚本并用 `sh kernel-chktaint` 执行，它会在上面引用的日志中有类似语句的机器上打印这样的内容：

```
Kernel is Tainted for following reasons:
* Proprietary module was loaded (#0)
* Kernel issued warning (#9)
* Externally-built ('out-of-tree') module was loaded (#12)
See Documentation/admin-guide/tainted-kernels.rst in the Linux kernel or
https://www.kernel.org/doc/html/latest/admin-guide/tainted-kernels.html for
a more details explanation of the various taint flags.
Raw taint value as int/string: 4609/'P      W 0      '
```

你也可以试着自己解码这个数字。如果内核被污染的原因只有一个，那么这很简单，在本例中您可以通过下表找到数字。如果你需要解码有多个原因的数字，因为它是一个位域（bitfield），其中每个位表示一个特定类型的污染的存在或不存在，最好让前面提到的脚本来处理。但是如果您需要快速看一下，可以使用这个 shell 命令来检查设置了哪些位：

```
$ for i in $(seq 18); do echo $((($i-1)) $((($cat /proc/sys/kernel/tainted)>>($i-1)&1));done
```

## 污染状态代码表

位	日志	数字	内核被污染的原因
0	G/P	1	已加载专用模块
1	/F	2	模块被强制加载
2	/S	4	内核运行在不合规范的系统上
3	/R	8	模块被强制卸载
4	/M	16	处理器报告了机器检测异常 (MCE)
5	/B	32	引用了错误的页或某些意外的页标志
6	/U	64	用户空间应用程序请求的污染
7	/D	128	内核最近死机了，即曾出现 OOPS 或 BUG
8	/A	256	ACPI 表被用户覆盖
9	/W	512	内核发出警告
10	/C	1024	已加载 staging 驱动程序
11	/I	2048	已应用平台固件缺陷的解决方案
12	/O	4096	已加载外部构建 (“树外”) 模块
13	/E	8192	已加载未签名的模块
14	/L	16384	发生软锁定
15	/K	32768	内核已实时打补丁
16	/X	65536	备用污染，为发行版定义并使用
17	/T	131072	内核是用结构随机化插件构建的

注：字符 \_ 表示空白，以便于阅读表。

## 污染的更详细解释

- 0) G 加载的所有模块都有 GPL 或兼容许可证，P 加载了任何专有模块。没有 MODULE\_LICENSE (模块许可证) 或 MODULE\_LICENSE 未被 insmod 认可为 GPL 兼容的模块被认为是专有的。
- 1) F 任何模块被 insmod -f 强制加载，' ' 所有模块正常加载。
- 2) S 内核运行在不合规范的处理器或系统上：硬件已运行在不受支持的配置中，因此无法保证正确执行。内核将被污染，例如：
  - 在 x86 上：PAE 是通过 intel CPU (如 Pentium M) 上的 forcepae 强制执行的，这些 CPU 不报告 PAE，但可能有功能实现，SMP 内核在非官方支持的 SMP Athlon CPU 上运行，MSR 被暴露到用户空间中。
  - 在 arm 上：在某些 CPU (如 Keystone 2) 上运行的内核，没有启用某些内核特性。

- 在 arm64 上：CPU 之间存在不匹配的硬件特性，引导加载程序以不同的模式引导 CPU。
- 某些驱动程序正在被用在不受支持的体系结构上（例如 x86\_64 以外的其他系统上的 scsi/snic，非 x86/x86\_64/itanium 上的 scsi/ips，已经损坏了 arm64 上 irqchip/irq-gic 的固件设置…）。

- 3) R 模块被 `rmmode -f` 强制卸载，' ' 所有模块都正常卸载。
- 4) M 任何处理器报告了机器检测异常，' ' 未发生机器检测异常。
- 5) B 页面释放函数发现错误的页面引用或某些意外的页面标志。这表示硬件问题或内核错误；日志中应该有其他信息指示发生此污染的原因。
- 6) U 用户或用户应用程序特意请求设置受污染标志，否则应为 ' '。
- 7) D 内核最近死机了，即出现了 OOPS 或 BUG。
- 8) A ACPI 表被重写。
- 9) W 内核之前已发出过警告（尽管有些警告可能会设置更具体的污染标志）。
- 10) C 已加载 staging 驱动程序。
- 11) I 内核正在处理平台固件（BIOS 或类似软件）中的严重错误。
- 12) O 已加载外部构建（“树外”）模块。
- 13) E 在支持模块签名的内核中加载了未签名的模块。
- 14) L 系统上先前发生过软锁定。
- 15) K 内核已经实时打了补丁。
- 16) X 备用污染，由 Linux 发行版定义和使用。
- 17) T 内核构建时使用了 `randstruct` 插件，它可以有意生成非常不寻常的内核结构布局（甚至是性能病态的布局），这在调试时非常有用。于构建时设置。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:<alexs@kernel.org>)。

---

**Original** [.../..../admin-guide/init](#)

译者 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

## 解释 “No working init found.” 启动挂起消息

作者 Andreas Mohr <andi at lisas period de>

Cristian Souza <cristianmsbr at gmail period com>

本文档提供了加载初始化二进制 (init binary) 失败的一些高级原因 (大致按执行顺序列出)。

- 1) 无法挂载根文件系统 **Unable to mount root FS** : 请设置“debug”内核参数 (在引导加载程序 bootloader 配置文件或 CONFIG\_CMDLINE) 以获取更详细的内核消息。
- 2) 初始化二进制不存在于根文件系统上 **init binary doesn't exist on rootfs** : 确保您的根文件系统类型正确 (并且 root= 内核参数指向正确的分区); 拥有所需的驱动程序, 例如 SCSI 或 USB 等存储硬件; 文件系统 (ext3、jffs2 等) 是内建的 (或者作为模块由 initrd 预加载)。
- 3) 控制台设备损坏 **Broken console device** : console= setup 中可能存在冲突 -> 初始控制台不可用 (initial console unavailable)。例如, 由于串行 IRQ 问题 (如缺少基于中断的配置) 导致的某些串行控制台不可靠。尝试使用不同的 console= device 或像 netconsole=。
- 4) 二进制存在但依赖项不可用 **Binary exists but dependencies not available** : 例如初始化二进制的必需库依赖项, 像 /lib/ld-linux.so.2 丢失或损坏。使用 readelf -d <INIT>|grep NEEDED 找出需要哪些库。
- 5) 无法加载二进制 **Binary cannot be loaded** : 请确保二进制的体系结构与您的硬件匹配。例如 i386 不匹配 x86\_64, 或者尝试在 ARM 硬件上加载 x86。如果您尝试在此处加载非二进制文件 (shell 脚本?), 您应该确保脚本在其工作头 (shebang header) 行 #!/... 中指定能正常工作的解释器 (包括其库依赖项)。在处理脚本之前, 最好先测试一个简单的非脚本二进制文件, 比如 /bin/sh, 并确认它能成功执行。要了解更多信息, 请将代码添加到 init/main.c 以显示 kernel\_execve() 的返回值。

当您发现新的失败原因时, 请扩展本解释 (毕竟加载初始化二进制是一个关键且艰难的过渡步骤, 需要尽可能无痛地进行), 然后向 LKML 提交一个补丁。

待办事项:

- 通过一个可以存储 kernel\_execve() 结果值的结构体数组实现各种 run\_init\_process() 调用, 并在失败时通过迭代 所有结果来记录一切 (非常重要的可用性修复)。
- 试着使实现本身在一般情况下更有帮助, 例如在受影响的地方提供额外的错误消息。

Todolist:

- reporting-bugs
- ramoops
- dynamic-debug-howto
- kdump/index
- perf/index

这是应用程序开发人员感兴趣的章节的开始。可以在这里找到涵盖内核 ABI 各个方面的文档。

Todolist:

- sysfs-rules

本手册的其余部分包括各种指南，介绍如何根据您的喜好配置内核的特定行为。

### 清除 **WARN\_ONCE**

`WARN_ONCE` / `WARN_ON_ONCE` / `printk_once` 仅仅打印一次消息。

```
echo 1 > /sys/kernel/debug/clear_warn_once
```

可以清除这种状态并且再次允许打印一次告警信息，这对于运行测试集后重现问题很有用。

### CPU 负载

Linux 通过``/proc/stat``和``/proc/uptime``导出各种信息，用户空间工具如 `top(1)` 使用这些信息计算系统花费在某个特定状态的平均时间。例如：

```
$ iostat Linux 2.6.18.3-exp (linmac) 02/20/2007
```

```
avg-cpu: %user %nice %system %iowait %steal %idle 10.01 0.00 2.92 5.44 0.00  
81.63
```

```
...
```

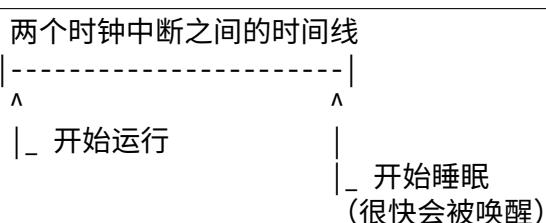
这里系统认为在默认采样周期內有 10.01% 的时间工作在用户空间，2.92% 的时间用在系统空间，总体上有 81.63% 的时间是空闲的。

大多数情况下``/proc/stat``的信息几乎真实反映了系统信息，然而，由于内核采集这些数据的方式/时间的特点，有时这些信息根本不可靠。

那么这些信息是如何被搜集的呢？每当时间中断触发时，内核查看此刻运行的进程类型，并增加与此类型/状态进程对应的计数器的值。这种方法的问题是在两次时间中断之间系统（进程）能够在多种状态之间切换多次，而计数器只增加最后一种状态下的计数。

举例—

假设系统有一个进程以如下方式周期性地占用 cpu：



在上面的情况下，根据``/proc/stat``的信息（由于当系统处于空闲状态时，时间中断经常会发生）系统的负载将会是 0

大家能够想象内核的这种行为会发生在许多情况下，这将导致``/proc/stat`` 中存在相当古怪的信息：

```

/* gcc -o hog smallhog.c */
#include <time.h>
#include <limits.h>
#include <signal.h>
#include <sys/time.h>
#define HIST 10

static volatile sig_atomic_t stop;

static void sighandler (int signr)
{
(void) signr;
stop = 1;
}
static unsigned long hog (unsigned long niters)
{
stop = 0;
while (!stop && --niters);
return niters;
}
int main (void)
{
int i;
struct itimerval it = { .it_interval = { .tv_sec = 0, .tv_usec = 1 },
                        .it_value = { .tv_sec = 0, .tv_usec = 1 } };
sigset(SIGALRM, &sighandler);
setitimer (ITIMER_REAL, &it, NULL);

hog (ULONG_MAX);
for (i = 0; i < HIST; ++i) v[i] = ULONG_MAX - hog (ULONG_MAX);
for (i = 0; i < HIST; ++i) tmp += v[i];
tmp /= HIST;
n = tmp - (tmp / 3.0);

sigemptyset (&set);
sigaddset (&set, SIGALRM);

for (;;) {
    hog (n);
    sigwait (&set, &i);
}
return 0;
}

```

参考—

- <https://lore.kernel.org/r/loom.20070212T063225-663@post.gmane.org>

- Documentation/filesystems/proc.rst (1.8)

谢谢—

Con Kolivas, Pavel Machek

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/admin-guide/cputopology.rst

翻译 唐艺舟 Tang Yizhou <[tangyeechou@gmail.com](mailto:tangyeechou@gmail.com)>

### 如何通过 sysfs 将 CPU 拓扑导出

CPU 拓扑信息通过 sysfs 导出。显示的项（属性）和某些架构的/proc/cpuinfo 输出相似。它们位于 /sys/devices/system/cpu/cpuX/topology/。请阅读 ABI 文件：Documentation/ABI/stable/sysfs-devices-system-cpu。

drivers/base/topology.c 是体系结构中性的，它导出了这些属性。然而，die、cluster、book、draw 这些层次结构相关的文件仅在体系结构提供了下文描述的宏的条件下被创建。

对于支持这个特性的体系结构，它必须在 include/asm-XXX/topology.h 中定义这些宏中的一部分：

```
#define topology_physical_package_id(cpu)
#define topology_die_id(cpu)
#define topology_cluster_id(cpu)
#define topology_core_id(cpu)
#define topology_book_id(cpu)
#define topology_drawer_id(cpu)
#define topology_sibling_cpumask(cpu)
#define topology_core_cpumask(cpu)
#define topology_cluster_cpumask(cpu)
#define topology_die_cpumask(cpu)
#define topology_book_cpumask(cpu)
#define topology_drawer_cpumask(cpu)
```

\*\*\_id macros 的类型是 int。\*\*\_cpumask macros 的类型是 (const) struct cpumask \*。后者和恰当的\*\*\_siblings sysfs 属性对应（除了 topology\_sibling\_cpumask()，它和 thread\_siblings 对应）。

为了在所有体系结构上保持一致，include/linux/topology.h 提供了上述所有宏的默认定义，以防它们未在 include/asm-XXX/topology.h 中定义：

- 1) topology\_physical\_package\_id: -1
- 2) topology\_die\_id: -1
- 3) topology\_cluster\_id: -1
- 4) topology\_core\_id: 0
- 5) topology\_book\_id: -1
- 6) topology\_drawer\_id: -1
- 7) topology\_sibling\_cpumask: 仅入参 CPU
- 8) topology\_core\_cpumask: 仅入参 CPU
- 9) topology\_cluster\_cpumask: 仅入参 CPU
- 10) topology\_die\_cpumask: 仅入参 CPU
- 11) topology\_book\_cpumask: 仅入参 CPU
- 12) topology\_drawer\_cpumask: 仅入参 CPU

此外，CPU 拓扑信息由/sys/devices/system/cpu 提供，包含下述文件。输出对应的内部数据源放在方括号 (“[]”) 中。

<code>kernel_max:</code>	内核配置允许的最大 CPU 下标值。[NR_CPUS-1]
<code>offline:</code>	由于热插拔移除或者超过内核允许的 CPU 上限（上文描述的 <code>kernel_max</code> ）导致未上线的 CPU。[~cpu_online_mask + cpus >= NR_CPUS]
<code>online:</code>	在线的 CPU，可供调度使用。[cpu_online_mask]
<code>possible:</code>	已被分配资源的 CPU，如果它们 CPU 实际存在，可以上线。 [cpu_possible_mask]
<code>present:</code>	被系统识别实际存在的 CPU。[cpu_present_mask]

上述输出的格式和 cpulist\_parse() 兼容 [参见 <linux/cpumask.h>]。下面给些例子。

在本例中，系统中有 64 个 CPU，但是 CPU 32-63 超过了 `kernel_max` 值，因为 `NR_CPUS` 配置项是 32，取值范围被限制为 0..31。此外注意 CPU2 和 4-31 未上线，但是可以上线，因为它们同时存在于 `present` 和 `possible`：

```
kernel_max: 31
offline: 2,4-31,32-63
online: 0-1,3
possible: 0-31
present: 0-31
```

在本例中，`NR_CPUS` 配置项是 128，但内核启动时设置 `possible_cpus=144`。系统中有 4 个 CPU，CPU2 被手动设置下线（也是唯一一个可以上线的 CPU）：

```
kernel_max: 127
offline: 2,4-127,128-143
online: 0-1,3
possible: 0-127
present: 0-3
```

阅读 Documentation/core-api/cpu\_hotplug.rst 可了解开机参数 possible\_cpus=NUM，同时还可以了解各种 cpumask 的信息。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/admin-guide/lockup-watchdogs.rst

**Translator** Hailong Liu <[liu.hailong6@zte.com.cn](mailto:liu.hailong6@zte.com.cn)>

### Softlockup 与 hardlockup 检测机制 (又名:nmi\_watchdog)

Linux 中内核实现了一种用以检测系统发生 softlockup 和 hardlockup 的看门狗机制。

Softlockup 是一种会引发系统在内核态中一直循环超过 20 秒(详见下面“实现”小节)导致其他任务没有机会得到运行的 BUG。一旦检测到’ softlockup ’发生，默认情况下系统会打印当前堆栈跟踪信息并进入锁定状态。也可配置使其在检测到’ softlockup ’后进入 panic 状态；通过 sysctl 命令设置 “kernel.softlockup\_panic”、使用内核启动参数 “softlockup\_panic” (详见 Documentation/admin-guide/kernel-parameters.rst) 以及使能内核编译选项 “BOOTPARAM\_SOFTLOCKUP\_PANIC” 都可实现这种配置。

而’ hardlockup ’是一种会引发系统在内核态一直循环超过 10 秒钟 (详见“实现”小节) 导致其他中断没有机会运行的缺陷。与’ softlockup ’情况类似，除了使用 sysctl 命令设置 ‘hardlockup\_panic’、使能内核选项 “BOOTPARAM\_HARDLOCKUP\_PANIC” 以及使用内核参数 “nmi\_watchdog” (详见：“Documentation/admin-guide/kernel-parameters.rst”) 外，一旦检测到’ hardlockup ’默认情况下系统打印当前堆栈跟踪信息，然后进入锁定状态。

这个 panic 选项也可以与 panic\_timeout 结合使用 (这个 panic\_timeout 是通过稍具迷惑性的 sysctl 命令 “kernel.panic” 来设置)，使系统在 panic 指定时间后自动重启。

## 实现

Softlockup 和 hardlockup 分别建立在 hrtimer(高精度定时器) 和 perf 两个子系统上而实现。这也就意味着理论上任何架构只要实现了这两个子系统就支持这两种检测机制。

Hrtimer 用于周期性产生中断并唤醒 watchdog 线程；NMI perf 事件则以” watchdog\_thresh “(编译时默认初始化为 10 秒，也可通过” watchdog\_thresh “这个 sysctl 接口来进行配置修改) 为间隔周期产生以检测 hardlockups。如果一个 CPU 在这个时间段内没有检测到 hrtimer 中断发生，’ hardlockup 检测器’ (即 NMI perf 事件处理函数) 将会视系统配置而选择产生内核警告或者直接 panic。

而 watchdog 线程本质上是一个高优先级内核线程，每调度一次就对时间戳进行一次更新。如果时间戳在  $2 \times \text{watchdog\_thresh}$ (这个是 softlockup 的触发门限) 这段时间都未更新，那么 “softlockup 检测器” (内部 hrtimer 定时器回调函数) 会将相关的调试信息打印到系统日志中，然后如果系统配置了进入 panic 流程则进入 panic，否则内核继续执行。

Hrtimer 定时器的周期是  $2 \times \text{watchdog\_thresh}/5$ ，也就是说在 hardlockup 被触发前 hrtimer 有 2~3 次机会产生时钟中断。

如上所述，内核相当于为系统管理员提供了一个可调节 hrtimer 定时器和 perf 事件周期长度的调节旋钮。如何通过这个旋钮为特定使用场景配置一个合理的周期值要对 lockups 检测的响应速度和 lockups 检测开销这两者之间进行权衡。

默认情况下所有在线 cpu 上都会运行一个 watchdog 线程。不过在内核配置了” NO\_HZ\_FULL “的情况下 watchdog 线程默认只会运行在管家 (housekeeping)cpu 上，而” nohz\_full “启动参数指定的 cpu 上则不会有 watchdog 线程运行。试想，如果我们允许 watchdog 线程在” nohz\_full “指定的 cpu 上运行，这些 cpu 上必须得运行时钟定时器来激发 watchdog 线程调度；这样一来就会使” nohz\_full “保护用户程序免受内核干扰的功能失效。当然，副作用就是” nohz\_full “指定的 cpu 即使在内核产生了 lockup 问题我们也无法检测到。不过，至少我们可以允许 watchdog 线程在管家 (non-tickless) 核上继续运行以便我们能继续正常的监测这些 cpus 上的 lockups 事件。

不论哪种情况都可以通过 sysctl 命令 kernel.watchdog\_cpumask 来对没有运行 watchdog 线程的 cpu 集合进行调节。对于 nohz\_full 而言，如果 nohz\_full cpu 上有异常挂住的情况，通过这种方式打开这些 cpu 上的 watchdog 进行调试可能会有所作用。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/admin-guide/unicode.rst

译者 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

## Unicode (统一码) 支持

(英文版) 上次更新: 2005-01-17, 版本号 1.4

此文档由 H. Peter Anvin <[unicode@lanana.org](mailto:unicode@lanana.org)> 管理, 是 Linux 注册名称与编号管理局 (Linux Assigned Names And Numbers Authority, LANANA) 项目的一部分。现行版本请见:

<http://www.lanana.org/docs/unicode/admin-guide/unicode.rst>

## 简介

Linux 内核代码已被重写以使用 Unicode 来将字符映射到字体。下载一个 Unicode 到字体 (Unicode-to-font) 表, 八位字符集与 UTF-8 模式都将改用此字体来显示。

这微妙地改变了八位字符表的语义。现在的四个字符表是:

映射代号	映射名称	Escape 代码 (G0)
LAT1_MAP	Latin-1 (ISO 8859-1)	ESC ( B
GRAF_MAP	DEC VT100 pseudographics	ESC ( 0
IBMPC_MAP	IBM code page 437	ESC ( U
USER_MAP	User defined	ESC ( K

特别是 ESC ( U 不再是“直通字体”，因为字体可能与 IBM 字符集完全不同。例如，即使加载了一个 Latin-1 字体，也允许使用块图形 (block graphics)。

请注意，尽管这些代码与 ISO 2022 类似，但这些代码及其用途都与 ISO 2022 不匹配；Linux 有两个八位代码 (G0 和 G1)，而 ISO 2022 有四个七位代码 (G0-G3)。

根据 Unicode 标准/ISO 10646, U+F000 到 U+F8FF 被保留用于操作系统范围内的分配 (Unicode 标准将其称为“团体区域 (Corporate Zone)”，因为这对于 Linux 是不准确的，所以我们称之为“Linux 区域”)。选择 U+F000 作为起点，因为它允许直接映射区域以 2 的大倍数开始 (以防需要 1024 或 2048 个字符的字体)。这就留下 U+E000 到 U+EFFF 作为最终用户区。

[v1.2]: Unicodes 范围从 U+F000 到 U+F7FF 已经被硬编码为直接映射到加载的字体，绕过了翻译表。用户定义的映射现在默认为 U+F000 到 U+F0FF，模拟前述行为。实际上，此范围可能较短；例如，vgacon 只能处理 256 字符 (U+F000..U+F0FF) 或 512 字符 (U+F000..U+F1FF) 字体。

## Linux 区域中定义的实际字符

此外，还定义了 Unicode 1.1.4 中不存在的以下字符；这些字符由 DEC VT 图形映射使用。[v1.2] 此用法已过时，不应再使用；请参见下文。

U+F800	DEC VT GRAPHICS HORIZONTAL LINE SCAN 1
U+F801	DEC VT GRAPHICS HORIZONTAL LINE SCAN 3
U+F803	DEC VT GRAPHICS HORIZONTAL LINE SCAN 7
U+F804	DEC VT GRAPHICS HORIZONTAL LINE SCAN 9

DEC VT220 使用 6x10 字符矩阵，这些字符在 DEC VT 图形字符集中形成一个平滑的过渡。我省略了扫描 5 行，因为它也被用作块图形字符，因此被编码为 U+2500 FORMS LIGHT HORIZONTAL。

[v1.3]: 这些字符已正式添加到 Unicode 3.2.0 中；它们在 U+23BA、U+23BB、U+23BC、U+23BD 处添加。Linux 现在使用新值。

[v1.2]: 添加了以下字符来表示常见的键盘符号，这些符号不太可能被添加到 Unicode 中，因为它们非常讨厌地取决于特定供应商。当然，这是糟糕设计的一个好例子。

U+F810	KEYBOARD SYMBOL FLYING FLAG
U+F811	KEYBOARD SYMBOL PULLDOWN MENU
U+F812	KEYBOARD SYMBOL OPEN APPLE
U+F813	KEYBOARD SYMBOL SOLID APPLE

## 克林贡 (Klingon) 语支持

1996 年，Linux 是世界上第一个添加对人工语言克林贡支持的操作系统，克林贡是由 Marc Okrand 为《星际迷航》电视连续剧创造的。这种编码后来被征募 Unicode 注册表 (ConScript Unicode Registry, CSUR) 采用，并建议（但最终被拒绝）纳入 Unicode 平面一。不过，它仍然是 Linux 区域中的 Linux/CSUR 私有分配。

这种编码已经得到克林贡语言研究所 (Klingon Language Institute) 的认可。有关更多信息，请联系他们：

<http://www.kli.org/>

由于 Linux CZ 开头部分的字符大多是 dingbats/symbols/forms 类型，而且这是一种语言，因此根据标准 Unicode 惯例，我将它放置在 16 单元的边界上。

---

**Note:** 这个范围现在由征募 Unicode 注册表正式管理。规范性引用文件为：

<https://www.evertype.com/standards/csur/klingon.html>

---

克林贡语有一个 26 个字符的字母表，一个 10 位数的位置数字书写系统，从左到右，从上到下书写。

克林贡字母的几种字形已经被提出。但是由于这组符号看起来始终是一致的，只有实际的形状不同，因此按照标准 Unicode 惯例，这些差异被认为是字体变体。

U+F8D0	KLINGON LETTER A
U+F8D1	KLINGON LETTER B
U+F8D2	KLINGON LETTER CH
U+F8D3	KLINGON LETTER D
U+F8D4	KLINGON LETTER E
U+F8D5	KLINGON LETTER GH
U+F8D6	KLINGON LETTER H
U+F8D7	KLINGON LETTER I
U+F8D8	KLINGON LETTER J
U+F8D9	KLINGON LETTER L
U+F8DA	KLINGON LETTER M
U+F8DB	KLINGON LETTER N
U+F8DC	KLINGON LETTER NG
U+F8DD	KLINGON LETTER O
U+F8DE	KLINGON LETTER P
U+F8DF	KLINGON LETTER Q - Written <q> in standard Okrand Latin transliteration
U+F8E0	KLINGON LETTER QH - Written <Q> in standard Okrand Latin transliteration
U+F8E1	KLINGON LETTER R
U+F8E2	KLINGON LETTER S
U+F8E3	KLINGON LETTER T
U+F8E4	KLINGON LETTER TLH
U+F8E5	KLINGON LETTER U
U+F8E6	KLINGON LETTER V
U+F8E7	KLINGON LETTER W
U+F8E8	KLINGON LETTER Y
U+F8E9	KLINGON LETTER GLOTTAL STOP
U+F8F0	KLINGON DIGIT ZERO
U+F8F1	KLINGON DIGIT ONE
U+F8F2	KLINGON DIGIT TWO
U+F8F3	KLINGON DIGIT THREE
U+F8F4	KLINGON DIGIT FOUR
U+F8F5	KLINGON DIGIT FIVE
U+F8F6	KLINGON DIGIT SIX
U+F8F7	KLINGON DIGIT SEVEN
U+F8F8	KLINGON DIGIT EIGHT
U+F8F9	KLINGON DIGIT NINE
U+F8FD	KLINGON COMMA
U+F8FE	KLINGON FULL STOP
U+F8FF	KLINGON SYMBOL FOR EMPIRE

## 其他虚构和人工字母

自从分配了克林贡 Linux Unicode 块之后，John Cowan <jcowan@reutershealth.com> 和 Michael Everson <everson@everttype.com> 建立了一个虚构和人工字母的注册表。征募 Unicode 注册表请访问：

<https://www.everttype.com/standards/csur/>

所使用的范围位于最终用户区域的低端，因此无法进行规范化分配，但建议希望对虚构字母进行编码的人员使用这些代码，以实现互操作性。对于克林贡语，CSUR 采用了 Linux 编码。CSUR 的人正在推动将 Tengwar 和 Cirth 添加到 Unicode 平面一；将克林贡添加到 Unicode 平面一被拒绝，因此上述编码仍然是官方的。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/admin-guide/sysrq.rst

翻译 黄军华 Junhua Huang <[huang.junhua@zte.com.cn](mailto:huang.junhua@zte.com.cn)>

校译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## Linux 魔法系统请求键骇客

针对 sysrq.c 的文档说明

### 什么是魔法 SysRq 键？

它是一个你可以输入的具有魔法般的组合键。无论内核在做什么，内核都会响应 SysRq 键的输入，除非内核完全卡死。

### 如何使能魔法 SysRq 键？

在配置内核时，我们需要设置 ‘Magic SysRq key (CONFIG\_MAGIC\_SYSRQ)’ 为 ‘Y’。当运行一个编译进 sysrq 功能的内核时，/proc/sys/kernel/sysrq 控制着被 SysRq 键调用的功能许可。这个文件的默认值由 CONFIG\_MAGIC\_SYSRQ\_DEFAULT\_ENABLE 配置符号设定，文件本身默认设置为 1。以下是 /proc/sys/kernel/sysrq 中可能的值列表：

- 0 - 完全不使能 SysRq 键

- 1 - 使能 SysRq 键的全部功能
- >1 - 对于允许的 SysRq 键功能的比特掩码（参见下面更详细的功能描述）：

2 = 0x2 - 使能对控制台日志记录级别的控制
4 = 0x4 - 使能对键盘的控制 (SAK, unraw)
8 = 0x8 - 使能对进程的调试导出等
16 = 0x10 - 使能同步命令
32 = 0x20 - 使能重新挂载只读
64 = 0x40 - 使能对进程的信号操作 (term, kill, oom-kill)
128 = 0x80 - 允许重启、断电
256 = 0x100 - 允许让所有实时任务变普通任务

你可以通过如下命令把值设置到这个文件中：

```
echo "number" >/proc/sys/kernel/sysrq
```

这里被写入的 number 可以是 10 进制数，或者是带着 0x 前缀的 16 进制数。CONFIG\_MAGIC\_SYSRQ\_DEFAULT\_ENABLE 必须是以 16 进制数写入。

注意，/proc/sys/kernel/sysrq 的值只影响通过键盘触发 SysRq 的调用，对于通过 /proc/sysrq-trigger 的任何操作调用都是允许的（通过具有系统权限的用户）。

### 如何使用魔法 SysRq 键？

在 **x86** 架构上 你可以按下键盘组合键 ALT-SysRq-<command key>。

---

**Note:** 一些键盘可能没有标识 ‘SysRq’ 键。‘SysRq’ 键也被当做 ‘Print Screen’ 键。同时有些键盘无法处理同时按下这么多键，因此你可以先按下键盘 Alt 键，然后按下键盘 SysRq 键，再释放键盘 SysRq 键，之后按下键盘上命令键 <command key>，最后释放所有键。

---

在 **SPARC** 架构上 你可以按下键盘组合键 ALT-STOP-<command key>。

在串行控制台（只针对 **PC** 类型的标准串口） 你可以发一个 BREAK，然后在 5 秒内发送一个命令键，发送 BREAK 两次将被翻译为一个正常的 BREAK 操作。

在 **PowerPC** 架构上 按下键盘组合键 ALT - Print Screen （或者 F13） - < 命令键 >。Print Screen （或者 F13） - < 命令键 > 或许也能实现。

在其他架构上 如果你知道其他架构的组合键，请告诉我，我可以把它们添加到这部分。

在所有架构上 写一个字符到 /proc/sysrq-trigger 文件，例如：

```
echo t > /proc/sysrq-trigger
```

这个命令键 <command key> 是区分大小写的。

## 什么是命令键?

命令键	功能
b	将立即重启系统，不会同步或者卸载磁盘。
c	将执行系统 crash，如果配置了系统 crashdump，将执行 crashdump。
d	显示所有持有的锁。
e	发送 SIGTERM 信号给所有进程，除了 init 进程。
f	将调用 oom killer 杀掉一个过度占用内存的进程，如果什么任务都没杀，也不会 panic。
g	kgdb 使用（内核调试器）。
h	将会显示帮助。（实际上除了这里列举的键，其他的都将显示帮助，但是 h 容易记住）:-)
i	发送 SIGKILL 给所有进程，除了 init 进程。
j	强制性的“解冻它” - 用于被 FIFREEZE ioctl 操作冻住的文件系统。
k	安全访问秘钥 (SAK) 杀掉在当前虚拟控制台的所有程序，注意：参考下面 SAK 节重要论述。
l	显示所有活动 cpu 的栈回溯。
m	将导出当前内存信息到你的控制台。
n	用于使所有实时任务变成普通任务。
o	将关闭系统（如果配置和支持的话）。
p	将导出当前寄存器和标志位到控制台。
q	将导出每个 cpu 上所有已装备的高精度定时器（不是完整的 time_list 文件显示的 timers）和所有时钟事件设备的详细信息。
r	关闭键盘的原始模式，设置为转换模式。
s	将尝试同步所有的已挂载文件系统。
t	将导出当前所有任务列表和它们的信息到控制台。
u	将尝试重新挂载已挂载文件系统为只读。
v	强制恢复帧缓存控制台。
v	触发 ETM 缓存导出 [ARM 架构特有]
w	导出处于不可中断状态（阻塞）的任务。
x	在 ppc/powerpc 架构上用于 xmon 接口。在 sparc64 架构上用于显示全局的 PMU（性能监控单元）寄存器。在 MIPS 架构上导出所有的 tlb 条目。
y	显示全局 cpu 寄存器 [SPARC-64 架构特有]
z	导出 ftrace 缓存信息
0-9	设置控制台日志级别，该级别控制什么样的内核信息将被打印到你的控制台。（比如 0，将使得只有紧急信息，像 PANICs or OOPSes 才能到你的控制台。）

### 好了，我能用他们做什么呢？

嗯，当你的 X 服务端或者 svgalib 程序崩溃，unraw(r) 非原始模式命令键是非常方便的。

sak(k)（安全访问秘钥）在你尝试登陆的同时，又想确保当前控制台没有可以获取你的密码的特洛伊木马程序运行时是有用的。它会杀掉给定控制台的所有程序，这样你就可以确认当前的登陆提示程序是实际来自 init 进程的程序，而不是某些特洛伊木马程序。

---

**Important:** 在其实际的形式中，在兼容 C2 安全标准的系统上，它不是一个真正的 SAK，它也不应该误认为此。

---

似乎其他人发现其可以作为（系统终端联机键）当你想退出一个程序，同时不会让你切换控制台的方法。（比如，X 服务端或者 svgalib 程序）

reboot(b) 是个好方法，当你不能关闭机器时，它等同于按下“复位”按钮。

crash(c) 可以用于手动触发一个 crashdump，当系统卡住时。注意当 crashdump 机制不可用时，这个只是触发一个内核 crash。

sync(s) 在拔掉可移动介质之前，或者在使用不提供优雅关机的救援 shell 之后很方便 - 它将确保你的数据被安全地写入磁盘。注意，在你看到屏幕上出现“OK”和“Done”之前，同步还没有发生。

umount(u) 可以用来标记文件系统正常卸载，从正在运行的系统角度来看，它们将被重新挂载为只读。这个重新挂载动作直到你看到“OK”和“Done”信息出现在屏幕上才算完成。

日志级别 0 - 9 用于当你的控制台被大量的内核信息冲击，你不想看见的时候。选择 0 将禁止除了最紧急的内核信息外的所有的内核信息输出到控制台。（但是如果 syslogd/klogd 进程是运行的，它们仍将被记录。）

term(e) 和 kill(i) 用于当你有些有点失控的进程，你无法通过其他方式杀掉它们的时候，特别是它正在创建其他进程。

“just thaw it(j)” 用于当你的系统由于一个 FIFREEZE ioctl 调用而产生的文件系统冻结，而导致的不响应时。

### 有的时候 SysRq 键在使用它之后，看起来像是“卡住”了，我能做些什么？

这也会发生在我这，我发现轻敲键盘两侧的 shift、alt 和 control 键，然后再次敲击一个无效的 SysRq 键序列可以解决问题。（比如，像键盘组合键 alt-sysrq-z）切换到另一个虚拟控制台（键盘操作 ALT+Fn），然后再切回来应该也有帮助。

## 我敲击了 SysRq 键，但像是什么都没发生，发生了什么错误？

有一些键盘对于 SysRq 键设置了不同的键值，而不是提前定义的 99 (查看在 `include/uapi/linux/input-event-codes.h` 文件中 `KEY_SYSRQ` 的定义) 或者就根本没有 SysRq 键。在这些场景下，执行 `showkey -s` 命令来找到一个合适的扫描码序列，然后使用 `setkeycodes <sequence> 99` 命令映射这个序列值到通用的 SysRq 键编码上 (比如 `setkeycodes e05b 99`)。最好将这个命令放在启动脚本中。哦，顺便说一句，你十秒钟不输入任何东西就将退出 “`showkey`”。

## 我想添加一个 SysRq 键事件到一个模块中，如何去做呢？

为了注册一个基础函数到这个表中，首先你必须包含 `include/linux/sysrq.h` 头文件，这个头文件定义了你所需要的所有东西。然后你必须创建一个 `sysrq_key_op` 结构体，然后初始化它，使用如下内容，A) 你将使用的这个键的处理函数，B) 一个 `help_msg` 字符串，在 SysRq 键打印帮助信息时将打印出来，C) 一个 `action_msg` 字符串，就在你的处理函数调用前打印出来。你的处理函数必须符合在 ‘`sysrq.h`’ 文件中的函数原型。

在 `sysrq_key_op` 结构体被创建后，你可以调用内核函数 `register_sysrq_key(int key, const struct sysrq_key_op *op_p);`，该函数在表中的 ‘key’ 对应位置内容是空的情况下，将通过 `op_p` 指针注册这个操作函数到表中 ‘key’ 对应位置上。在模块卸载的时候，你必须调用 `unregister_sysrq_key(int key, const struct sysrq_key_op *op_p)` 函数，该函数只有在当前该键对应的处理函数被注册到了 ‘key’ 对应位置时，才会移除 ‘`op_p`’ 指针对应的键值操作函数。这是为了防止在你注册之后，该位置被改写的情况。

魔法 SysRq 键系统的工作原理是将键对应操作函数注册到键的操作查找表，该表定义在 ‘`drivers/tty/sysrq.c`’ 文件中。该键表有许多在编译时候就注册进去的操作函数，但是是可变的。并且有两个函数作为操作该表的接口被导出：

`register_sysrq_key` 和 `unregister_sysrq_key`.

当然，永远不要在表中留下无效指针，即，当你的模块存在调用 `register_sysrq_key()` 函数，它一定要调用 `unregister_sysrq_key()` 来清除它使用过的 SysRq 键表条目。表中的空指针是安全的。:)

如果对于某种原因，在 `handle_sysrq` 调用的处理函数中，你认为有必要调用 `handle_sysrq` 函数时，你必须意识到当前你处于一个锁中 (你也同时处在一个中断处理函数中，这意味着不能睡眠)。所以这时你必须使用 `_handle_sysrq_nolock` 替代。

## 当我敲击一个 SysRq 组合键时，只有标题打印出现在控制台？

SysRq 键的输出和所有其他控制台输出一样，受制于控制台日志级别控制。这意味着，如果内核以发行版内核中常见的 “quiet” 方式启动，则输出可能不会出现在实际的控制台上，即使它会出现在 `dmesg` 缓存中，也可以通过 `dmesg` 命令和 `/proc/kmsg` 文件的消费访问到。作为一个特例，来自 `sysrq` 命令的标题行将被传递给所有控制台使用者，就好像当前日志级别是最大的一样。如果只发出标题头，则几乎可以肯定内核日志级别太低。如果你需要控制台上的输出，那么你将需要临时提高控制台日志级别，通过使用键盘组合键 `alt-sysrq-8` 或者：

```
echo 8 > /proc/sysrq-trigger
```

在触发了你感兴趣的 SysRq 键命令后，记得恢复日志级别到正常情况。

### 我有很多问题时，可以请教谁？

请教在内核邮件列表上的人，邮箱：[linux-kernel@vger.kernel.org](mailto:linux-kernel@vger.kernel.org)

### 致谢

- Mydraal <[vulpyne@vulpyne.net](mailto:vulpyne@vulpyne.net)> 撰写了该文件
- Adam Sulmicki <[adam@cfar.umd.edu](mailto:adam@cfar.umd.edu)> 进行了更新
- Jeremy M. Dolan <[jmd@turbogeek.org](mailto:jmd@turbogeek.org)> 在 2001/01/28 10:15:59 进行了更新
- Crutcher Dunnivant <[crutcher+kernel@datastacks.com](mailto:crutcher+kernel@datastacks.com)> 添加键注册部分

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

---

**Original** Documentation/admin-guide/mm/index.rst

翻译 徐鑫 xu xin <[xu.xin16@zte.com.cn](mailto:xu.xin16@zte.com.cn)>

## 内存管理

Linux 内存管理子系统，顾名思义，是负责系统中的内存管理。它包括了虚拟内存与请求分页的实现，内核内部结构和用户空间程序的内存分配、将文件映射到进程地址空间以及许多其他很酷的事情。

Linux 内存管理是一个具有许多可配置设置的复杂系统，且这些设置中的大多数都可以通过 /proc 文件系统获得，并且可以使用 sysctl 进行查询和调整。这些 API 接口被描述在 Documentation/admin-guide/sysctl/vm.rst 文件和 [man 5 proc](#) 中。

Linux 内存管理有它自己的术语，如果你还不熟悉它，请考虑阅读下面参考：Documentation/admin-guide/mm/concepts.rst.

在此目录下，我们详细描述了如何与 Linux 内存管理中的各种机制交互。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/admin-guide/mm/damon/index.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译

## 监测数据访问

DAMON 允许轻量级的数据访问监测。使用 DAMON，用户可以分析他们系统的内存访问模式，并优化它们。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/admin-guide/mm/damon/start.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译

## 入门指南

本文通过演示 DAMON 的默认用户空间工具，简要地介绍了如何使用 DAMON。请注意，为了简洁起见，本文档只描述了它的部分功能。更多细节请参考该工具的使用文档。[doc](#) .

## 前提条件

内核

首先，你要确保你当前系统中跑的内核构建时选定了这个功能选项 `CONFIG_DAMON_*=y`。

用户空间工具

在演示中，我们将使用 DAMON 的默认用户空间工具，称为 DAMON Operator (DAMO)。它可以在 <https://github.com/awslabs/damo> 找到。下面的例子假设 DAMO 在你的 \$PATH 上。当然，但这并不是强制性的。

因为 DAMO 使用的是 DAMON 的 debugfs 接口 (详情请参考[详细用法](#) 中的使用方法) 你应该确保 debugfs 被挂载。手动挂载它，如下所示：

```
# mount -t debugfs none /sys/kernel/debug/
```

或者在你的 `/etc/fstab` 文件中添加以下一行，这样你的系统就可以在启动时自动挂载 `debugfs` 了：

```
debugfs /sys/kernel/debug debugfs defaults 0 0
```

## 记录数据访问模式

下面的命令记录了一个程序的内存访问模式，并将监测结果保存到文件中。

```
$ git clone https://github.com/sjp38/masim  
$ cd masim; make; ./masim ./configs/zigzag.cfg &  
$ sudo damo record -o damon.data $(pidof masim)
```

命令的前两行下载了一个人工内存访问生成器程序并在后台运行。生成器将重复地逐一访问两个 100 MiB 大小的内存区域。你可以用你的真实工作负载来代替它。最后一行要求 `damo` 将访问模式记录在 `damon.data` 文件中。

## 将记录的模式可视化

你可以在 heatmap 中直观地看到这种模式，显示哪个内存区域（X 轴）何时被访问（Y 轴）以及访问的频率（数字）。：

你也可以直观地看到工作集的大小分布，按大小排序。：

```
$ sudo damo report wss --range 0 101 10
# <percentile> <wss>
# target_id      18446632103789443072
# avr: 107.708 MiB
0          0 B |
10         95.328 MiB *****
20         95.332 MiB *****
30         95.340 MiB *****
40         95.387 MiB *****
50         95.387 MiB *****
60         95.398 MiB *****
70         95.398 MiB *****
80         95.504 MiB *****
90        190.703 MiB *****
100        196.875 MiB *****
```

在上述命令中使用 `--sortby` 选项，可以显示工作集的大小是如何按时间顺序变化的。：

```
$ sudo damo report wss --range 0 101 10 --sortby time
# <percentile> <wss>
# target_id      18446632103789443072
# avr: 107.708 MiB
0      3.051 MiB |
10     190.703 MiB | *****
20     95.336 MiB | *****
30     95.328 MiB | *****
40     95.387 MiB | *****
50     95.332 MiB | *****
60     95.320 MiB | *****
70     95.398 MiB | *****
80     95.398 MiB | *****
90     95.340 MiB | *****
100    95.398 MiB | *****
```

### 数据访问模式感知的内存管理

以下三个命令使每一个大小  $>=4K$  的内存区域在你的工作负载中没有被访问  $>=60$  秒，就会被换掉。

```
$ echo "#min-size max-size min-acc max-acc min-age max-age action" > test_scheme
$ echo "4K      max      0      0      60s    max      pageout" >> test_scheme
$ damon schemes -c test_scheme <pid of your workload>
```

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:<alexs@kernel.org>)。

**Original** Documentation/admin-guide/mm/damon/usage.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译

### 详细用法

DAMON 为不同的用户提供了下面这些接口。

- DAMON 用户空间工具。这为有这特权的人，如系统管理员，希望有一个刚好可以工作的人性化界面。使用它，用户可以以人性化的方式使用 DAMON 的主要功能。不过，它可能不会为特殊情况进行高度调整。它同时支持虚拟和物理地址空间的监测。更多细节，请参考它的 [使用文档](#)。
- sysfs 接口。这是为那些希望更高级的使用 DAMON 的特权用户空间程序员准备的。使用它，用户可以通过读取和写入特殊的 sysfs 文件来使用 DAMON 的主要功能。因此，你可以编写和使用你个性化的 DAMON sysfs 包装程序，代替你读/写 sysfs 文件。[DAMON 用户空间工具](#) 就是这种程序的一个例子。它同时支持虚拟和物理地址空间的监测。注意，这个界面只提供简单的监测结果统计。对于详细的监测结果，DAMON 提供了一个`:ref: 跟踪点 <tracepoint>`。
- debugfs interface. 这几乎与`:ref:sysfs interface <sysfs_interface>` 接口相同。这将在下一个 LTS 内核发布后被移除，所以用户应该转移到 sysfs interface。
- 内核空间编程接口。这是为内核空间程序员准备的。使用它，用户可以通过为你编写内核空间的 DAMON 应用程序，最灵活有效地利用 DAMON 的每一个功能。你甚至可以为各种地址空间扩展 DAMON。详细情况请参考接口文件。

## sysfs 接口

DAMON 的 sysfs 接口是在定义 CONFIG\_DAMON\_SYSFS 时建立的。它在其 sysfs 目录下创建多个目录和文件, <sysfs>/kernel/mm/damon/。你可以通过对该目录下的文件进行写入和读取来控制 DAMON。

对于一个简短的例子，用户可以监测一个给定工作负载的虚拟地址空间，如下所示：

```
# cd /sys/kernel/mm/damon/admin/
# echo 1 > kdamonds/nr && echo 1 > kdamonds/0/contexts/nr
# echo vaddr > kdamonds/0/contexts/0/operations
# echo 1 > kdamonds/0/contexts/0/targets/nr
# echo $(pidof <workload>) > kdamonds/0/contexts/0/targets/0/pid
# echo on > kdamonds/0/state
```

## 文件层次结构

DAMON sysfs 接口的文件层次结构如下图所示。在下图中，父子关系用缩进表示，每个目录有 / 后缀，每个目录中的文件用逗号 (”,) 分开。

```
/sys/kernel/mm/damon/admin
  kdamonds/nr_kdamonds
    0/state,pid
      contexts/nr_contexts
        0/operations
          monitoring_attrs/
            intervals/sample_us,aggr_us,update_us
            nr_regions/min,max
          targets/nr_targets
            0/pid_target
              regions/nr_regions
                0/start,end
                ...
            ...
          schemes/nr_schemes
            0/action
              access_pattern/
                sz/min,max
                nr_accesses/min,max
                age/min,max
              quotas/ms,bytes,reset_interval_ms
                weights/sz_permil,nr_accesses_permil,age_permil
              watermarks/metric,interval_us,high,mid,low
              stats/nr_tried,sz_tried,nr_applied,sz_applied,qt_exceeds
            ...
    ...
  ...
...
```

### 根

DAMON sysfs 接口的根是 `<sysfs>/kernel/mm/damon/`，它有一个名为 `admin` 的目录。该目录包含特权用户空间程序控制 DAMON 的文件。拥有根权限的用户空间工具或 deamons 可以使用这个目录。

### **kdamonds/**

与监测相关的信息包括请求规格和结果被称为 DAMON 上下文。DAMON 用一个叫做 `kdamond` 的内核线程执行每个上下文，多个 `kdamonds` 可以并行运行。

在 `admin` 目录下，有一个目录，即```kdamonds```，它有控制 `kdamonds` 的文件存在。在开始时，这个目录只有一个文件，`nr_kdamonds`。向该文件写入一个数字 (`N`)，就会创建名为 0 到 `N-1` 的子目录数量。每个目录代表每个 `kdamond`。

### **kdamonds/<N>/**

在每个 `kdamond` 目录中，存在两个文件 (`state` 和 `pid`) 和一个目录 (`contexts`)。

读取 `state` 时，如果 `kdamond` 当前正在运行，则返回 `on`，如果没有运行则返回 `off`。写入 `on` 或 `off` 使 `kdamond` 处于状态。向 `state` 文件写 `update_schemes_stats`，更新 `kdamond` 的每个基于 DAMON 的操作方案的统计文件的内容。关于统计信息的细节，请参考 `stats section`.

如果状态为 `on`，读取 `pid` 显示 `kdamond` 线程的 `pid`。

`contexts` 目录包含控制这个 `kdamond` 要执行的监测上下文的文件。

### **kdamonds/<N>/contexts/**

在开始时，这个目录只有一个文件，即 `nr_contexts`。向该文件写入一个数字 (`N`)，就会创建名为``0``到 `N-1` 的子目录数量。每个目录代表每个监测背景。目前，每个 `kdamond` 只支持一个上下文，所以只有 0 或 1 可以被写入文件。

### **contexts/<N>/**

在每个上下文目录中，存在一个文件 (`operations`) 和三个目录 (`monitoring_attrs`, `targets`, 和 `schemes`)。

DAMON 支持多种类型的监测操作，包括对虚拟地址空间和物理地址空间的监测。你可以通过向文件中写入以下关键词之一，并从文件中读取，来设置和获取 DAMON 将为上下文使用何种类型的监测操作。

- `vaddr`: 监测特定进程的虚拟地址空间
- `paddr`: 监视系统的物理地址空间

## contexts/<N>/monitoring\_attrs/

用于指定监测属性的文件，包括所需的监测质量和效率，都在 `monitoring_attrs` 目录中。具体来说，这个目录下有两个目录，即 `intervals` 和 `nr_regions`。

在 `intervals` 目录下，存在 DAMON 的采样间隔 (`sample_us`)、聚集间隔 (`aggr_us`) 和更新间隔 (`update_us`) 三个文件。你可以通过写入和读出这些文件来设置和获取微秒级的值。

在 `nr_regions` 目录下，有两个文件分别用于 DAMON 监测区域的下限和上限 (`min` 和 `max`)，这两个文件控制着监测的开销。你可以通过向这些文件的写入和读出来设置和获取这些值。

关于间隔和监测区域范围的更多细节，请参考设计文件 (`/vm/damon/design`)。

## contexts/<N>/targets/

在开始时，这个目录只有一个文件 `nr_targets`。向该文件写入一个数字 (N)，就可以创建名为 0 到 N-1 的子目录的数量。每个目录代表每个监测目标。

### targets/<N>/

在每个目标目录中，存在一个文件 (`pid_target`) 和一个目录 (`regions`)。

如果你把 `vaddr` 写到 `contexts/<N>/operations` 中，每个目标应该是一个进程。你可以通过将进程的 `pid` 写到 `pid_target` 文件中来指定 DAMON 的进程。

### targets/<N>/regions

当使用 `vaddr` 监测操作集时 (`vaddr` 被写入 `contexts/<N>/operations` 文件)，DAMON 自动设置和更新监测目标区域，这样就可以覆盖目标进程的整个内存映射。然而，用户可能希望将初始监测区域设置为特定的地址范围。

相反，当使用 `paddr` 监测操作集时，DAMON 不会自动设置和更新监测目标区域 (`paddr` 被写入 `contexts/<N>/operations` 中)。因此，在这种情况下，用户应该自己设置监测目标区域。

在这种情况下，用户可以按照自己的意愿明确设置初始监测目标区域，将适当的值写入该目录下的文件。

开始时，这个目录只有一个文件，`nr_regions`。向该文件写入一个数字 (N)，就可以创建名为 0 到 N-1 的子目录。每个目录代表每个初始监测目标区域。

### **regions/<N>/**

在每个区域目录中，你会发现两个文件（`start` 和 `end`）。你可以通过向文件写入和从文件中读出，分别设置和获得初始监测目标区域的起始和结束地址。

### **contexts/<N>/schemes/**

对于一版的基于 DAMON 的数据访问感知的内存管理优化，用户通常希望系统对特定访问模式的内存区域应用内存管理操作。DAMON 从用户那里接收这种形式化的操作方案，并将这些方案应用于目标内存区域。用户可以通过读取和写入这个目录下的文件来获得和设置这些方案。

在开始时，这个目录只有一个文件，`nr_schemes`。向该文件写入一个数字 (`N`)，就可以创建名为``0``到``N-1``的子目录的数量。每个目录代表每个基于 DAMON 的操作方案。

### **schemes/<N>/**

在每个方案目录中，存在四个目录 (`access_pattern`, `quotas`, ``watermarks``, 和 `stats`) 和一个文件 (`action`)。

`action` 文件用于设置和获取你想应用于具有特定访问模式的内存区域的动作。可以写入文件和从文件中读取的关键词及其含义如下。

- `willneed`: 对有 `MADV_WILLNEED` 的区域调用 `madvise()`。
- `cold`: 对具有 `MADV_COLD` 的区域调用 `madvise()`。
- `pageout`: 为具有 `MADV_PAGEOUT` 的区域调用 `madvise()`。
- `hugepage`: 为带有 `MADV_HUGEPAGE` 的区域调用 `madvise()`。
- `nohugepage`: 为带有 `MADV_NOHUGEPAGE` 的区域调用 `madvise()`。
- `stat`: 什么都不做，只计算统计数据

### **schemes/<N>/access\_pattern/**

每个基于 DAMON 的操作方案的目标访问模式由三个范围构成，包括以字节为单位的区域大小、每个聚合区间的监测访问次数和区域年龄的聚合区间数。

在 `access_pattern` 目录下，存在三个目录 (`sz`, `nr_accesses`, 和 `age`)，每个目录有两个文件 (`min` 和 `max`)。你可以通过向 `sz`, `nr_accesses`, 和 `age` 目录下的 `min` 和 `max` 文件分别写入和读取来设置和获取给定方案的访问模式。

## schemes/<N>/quotas/

每个 动作的最佳 目标访问模式取决于工作负载，所以不容易找到。更糟糕的是，将某些动作的方案设置得过于激进会造成严重的开销。为了避免这种开销，用户可以为每个方案限制时间和大小配额。具体来说，用户可以要求 DAMON 尽量只使用特定的时间（时间配额）来应用行动，并且在给定的时间间隔（重置间隔）内，只对具有目标访问模式的内存区域应用行动，而不使用特定数量（大小配额）。

当预计超过配额限制时，DAMON 会根据 目标访问模式的大小、访问频率和年龄，对找到的内存区域进行优先排序。为了进行个性化的优先排序，用户可以为这三个属性设置权重。

在 quotas 目录下，存在三个文件 (`ms`, `bytes`, `reset_interval_ms`) 和一个目录 (`weights`)，其中有三个文件 (`sz_permil`, `nr_accesses_permil`, 和 `age_permil`)。

你可以设置以毫秒为单位的 时间配额，以字节为单位的 大小配额，以及以毫秒为单位的 重置间隔，分别向这三个文件写入数值。你还可以通过向 `weights` 目录下的三个文件写入数值来设置大小、访问频率和年龄的优先权，单位为千分之一。

## schemes/<N>/watermarks/

为了便于根据系统状态激活和停用每个方案，DAMON 提供了一个称为水位的功能。该功能接收五个值，称为 度量、间隔、高、中、低。度量值是指可以测量的系统度量值，如自由内存比率。如果系统的度量值 高于 memoent 的高值或 低于低值，则该方案被停用。如果该值低于 中，则该方案被激活。

在水位目录下，存在五个文件 (`metric`, `interval_us`, ``high``, `mid`, and `low`) 用于设置每个值。你可以通过向这些文件的写入来分别设置和获取这五个值。

可以写入 `metric` 文件的关键词和含义如下。

- `none`: 忽略水位
- `free_mem_rate`: 系统的自由内存率（千分比）。

`interval` 应以微秒为单位写入。

## schemes/<N>/stats/

DAMON 统计每个方案被尝试应用的区域的总数量和字节数，每个方案被成功应用的区域的两个数字，以及超过配额限制的总数量。这些统计数据可用于在线分析或调整方案。

可以通过读取 stats 目录下的文件 (`nr_tried`, `sz_tried`, `nr_applied`, `sz_applied`, 和 `qt_exceeds`) 分别检索这些统计数据。这些文件不是实时更新的，所以你应该要求 DAMON sysfs 接口通过在相关的 `kdamonds/<N>/state` 文件中写入一个特殊的关键字 `update_schemes_stats` 来更新统计信息的文件内容。

## 用例

下面的命令应用了一个方案：“如果一个大小为 [4KiB, 8KiB] 的内存区域在 [10, 20] 的聚合时间间隔内显示出每一个聚合时间间隔 [0, 5] 的访问量，请分页该区域。对于分页，每秒最多只能使用 10ms，而且每秒分页不能超过 1GiB。在这一限制下，首先分页出具有较长年龄的内存区域。另外，每 5 秒钟检查一次系统的可用内存率，当可用内存率低于 50% 时开始监测和分页，但如果可用内存率大于 60%，或低于 30%，则停止监测。”

```
# cd <sysfs>/kernel/mm/damon/admin
# # populate directories
# echo 1 > kdamonds/nr_kdamonds; echo 1 > kdamonds/0/contexts/nr_contexts;
# echo 1 > kdamonds/0/contexts/0/schemes/nr_schemes
# cd kdamonds/0/contexts/0/schemes/0
# # set the basic access pattern and the action
# echo 4096 > access_patterns/sz/min
# echo 8192 > access_patterns/sz/max
# echo 0 > access_patterns/nr_accesses/min
# echo 5 > access_patterns/nr_accesses/max
# echo 10 > access_patterns/age/min
# echo 20 > access_patterns/age/max
# echo pageout > action
# # set quotas
# echo 10 > quotas/ms
# echo $((1024*1024*1024)) > quotas/bytes
# echo 1000 > quotas/reset_interval_ms
# # set watermark
# echo free_mem_rate > watermarks/metric
# echo 5000000 > watermarks/interval_us
# echo 600 > watermarks/high
# echo 500 > watermarks/mid
# echo 300 > watermarks/low
```

请注意，我们强烈建议使用用户空间的工具，如 `damo`，而不是像上面那样手动读写文件。以上只是一个例子。

## debugfs 接口

DAMON 导出了八个文件，`attrs`, `target_ids`, `init_regions`, `schemes`, `monitor_on`, `kdamond_pid`, `mk_contexts` 和 `rm_contexts` under its debugfs directory, `<debugfs>/damon/`.

## 属性

用户可以通过读取和写入 `attrs` 文件获得和设置 采样间隔、聚集间隔、更新间隔以及监测目标区域的最小/最大数量。要详细了解监测属性，请参考 `:doc:/vm/damon/design`。例如，下面的命令将这些值设置为 5ms、100ms、1000ms、10 和 1000，然后再次检查：

```
# cd <debugfs>/damon
# echo 5000 100000 1000000 10 1000 > attrs
# cat attrs
5000 100000 1000000 10 1000
```

## 目标 ID

一些类型的地址空间支持多个监测目标。例如，虚拟内存地址空间的监测可以有多个进程作为监测目标。用户可以通过写入目标的相关 `id` 值来设置目标，并通过读取 `target_ids` 文件来获得当前目标的 `id`。在监测虚拟地址空间的情况下，这些值应该是监测目标进程的 `pid`。例如，下面的命令将 `pid` 为 42 和 4242 的进程设为监测目标，并再次检查：

```
# cd <debugfs>/damon
# echo 42 4242 > target_ids
# cat target_ids
42 4242
```

用户还可以通过在文件中写入一个特殊的关键字“`paddrn`”来监测系统的物理内存地址空间。因为物理地址空间监测不支持多个目标，读取文件会显示一个假值，即 42，如下图所示：

```
# cd <debugfs>/damon
# echo paddr > target_ids
# cat target_ids
42
```

请注意，设置目标 ID 并不启动监测。

## 初始监测目标区域

在虚拟地址空间监测的情况下，DAMON 自动设置和更新监测的目标区域，这样就可以覆盖目标进程的整个内存映射。然而，用户可能希望将监测区域限制在特定的地址范围内，如堆、栈或特定的文件映射区域。或者，一些用户可以知道他们工作负载的初始访问模式，因此希望为“自适应区域调整”设置最佳初始区域。

相比之下，DAMON 在物理内存监测的情况下不会自动设置和更新监测目标区域。因此，用户应该自己设置监测目标区域。

在这种情况下，用户可以通过在 `init_regions` 文件中写入适当的值，明确地设置他们想要的初始监测目标区域。输入的每一行应代表一个区域，形式如下：

```
<target_idx> <start_address> <end_address>
```

目标 `idx` 应该是 `target_ids` 文件中目标的索引，从 0 开始，区域应该按照地址顺序传递。例如，下面的命令将设置几个地址范围，1-100 和 100-200 作为 pid 42 的初始监测目标区域，这是 `target_ids` 中的第一个（索引 0），另外几个地址范围，20-40 和 50-100 作为 pid 4242 的地址，这是 `target_ids` 中的第二个（索引 1）：

```
# cd <debugfs>/damon
# cat target_ids
42 4242
# echo "0    1      100
        0    100    200
        1    20     40
        1    50     100" > init_regions
```

请注意，这只是设置了初始的监测目标区域。在虚拟内存监测的情况下，DAMON 会在一个更新间隔后自动更新区域的边界。因此，在这种情况下，如果用户不希望更新的话，应该把更新间隔设置得足够大。

## 方案

对于通常的基于 DAMON 的数据访问感知的内存管理优化，用户只是希望系统对特定访问模式的内存区域应用内存管理操作。DAMON 从用户那里接收这种形式化的操作方案，并将这些方案应用到目标进程中。

用户可以通过读取和写入 `scheme` `debugfs` 文件来获得和设置这些方案。读取该文件还可以显示每个方案的统计数据。在文件中，每一个方案都应该在每一行中以下列形式表示出来：

```
<target access pattern> <action> <quota> <watermarks>
```

你可以通过简单地在文件中写入一个空字符串来禁用方案。

## 目标访问模式

< 目标访问模式 > 是由三个范围构成的，形式如下：

```
min-size max-size min-acc max-acc min-age max-age
```

具体来说，区域大小的字节数 (`min-size` 和 `max-size`)，访问频率的每聚合区间的监测访问次数 (`min-acc` 和 `max-acc`)，区域年龄的聚合区间数 (`min-age` 和 `max-age`) 都被指定。请注意，这些范围是封闭区间。

## 动作

`<action>` 是一个预定义的内存管理动作的整数， DAMON 将应用于具有目标访问模式的区域。支持的数字和它们的含义如下：

- 0: Call ```madvise()``` for the region with ```MADV_WILLNEED```
- 1: Call ```madvise()``` for the region with ```MADV_COLD```
- 2: Call ```madvise()``` for the region with ```MADV_PAGEOUT```
- 3: Call ```madvise()``` for the region with ```MADV_HUGEPAGE```
- 4: Call ```madvise()``` for the region with ```MADV_NOHUGEPAGE```
- 5: Do nothing but count the statistics

## 配额

每个 动作的最佳 目标访问模式取决于工作负载，所以不容易找到。更糟糕的是，将某个动作的方案设置得过于激进会导致严重的开销。为了避免这种开销，用户可以通过下面表格中的 `<quota>` 来限制方案的时间和大小配额：

`<ms> <sz> <reset interval> <priority weights>`

这使得 DAMON 在 `<reset interval>` 毫秒内，尽量只用 `<ms>` 毫秒的时间对 目标访问模式的内存区域应用动作，并在 `<reset interval>` 内只对最多 `<sz>` 字节的内存区域应用动作。将 `<ms>` 和 `<sz>` 都设置为零，可以禁用配额限制。

当预计超过配额限制时，DAMON 会根据 目标访问模式的大小、访问频率和年龄，对发现的内存区域进行优先排序。为了实现个性化的优先级，用户可以在 < 优先级权重 > 中设置这三个属性的权重，具体形式如下：

`<size weight> <access frequency weight> <age weight>`

## 水位

有些方案需要根据系统特定指标的当前值来运行，如自由内存比率。对于这种情况，用户可以为该条件指定水位。：

`<metric> <check interval> <high mark> <middle mark> <low mark>`

`<metric>` 是一个预定义的整数，用于要检查的度量。支持的数字和它们的含义如下。

- 0: 忽视水位
- 1: 系统空闲内存率 (千分比)

每隔 < 检查间隔 > 微秒检查一次公制的值。

如果该值高于 < 高标 > 或低于 < 低标 >，该方案被停用。如果该值低于 < 中标 >，该方案将被激活。

## 统计数据

它还统计每个方案被尝试应用的区域的总数量和字节数，每个方案被成功应用的区域的两个数量，以及超过配额限制的总数量。这些统计数据可用于在线分析或调整方案。

统计数据可以通过读取方案文件来显示。读取该文件将显示你在每一行中输入的每个 方案，统计的五个数字将被加在每一行的末尾。

## 例子

下面的命令应用了一个方案：“如果一个大小为 [4KiB, 8KiB] 的内存区域在 [10, 20] 的聚合时间间隔内显示出每一个聚合时间间隔 [0, 5] 的访问量，请分页出该区域。对于分页，每秒最多只能使用 10ms，而且每秒分页不能超过 1GiB。在这一限制下，首先分页出具有较长年龄的内存区域。另外，每 5 秒钟检查一次系统的可用内存率，当可用内存率低于 50% 时开始监测和分页，但如果可用内存率大于 60%，或低于 30%，则停止监测”：

```
# cd <debugfs>/damon
# scheme="4096 8192 0 5    10 20    2" # target access pattern and action
# scheme+=" 10 $((1024*1024*1024)) 1000" # quotas
# scheme+=" 0 0 100"                      # prioritization weights
# scheme+=" 1 5000000 600 500 300"        # watermarks
# echo "$scheme" > schemes
```

## 开关

除非你明确地启动监测，否则如上所述的文件设置不会产生效果。你可以通过写入和读取 `monitor_on` 文件来启动、停止和检查监测的当前状态。写入 `on` 该文件可以启动对有属性的目标的监测。写入 `off` 该文件则停止这些目标。如果每个目标进程被终止，DAMON 也会停止。下面的示例命令开启、关闭和检查 DAMON 的状态：

```
# cd <debugfs>/damon
# echo on > monitor_on
# echo off > monitor_on
# cat monitor_on
off
```

请注意，当监测开启时，你不能写到上述的 `debugfs` 文件。如果你在 DAMON 运行时写到这些文件，将会返回一个错误代码，如 `-EBUSY`。

## 监测线程 PID

DAMON 通过一个叫做 `kdamond` 的内核线程来进行请求监测。你可以通过读取 `kdamond_pid` 文件获得该线程的 `pid`。当监测被关闭时，读取该文件不会返回任何信息：

```
# cd <debugfs>/damon
# cat monitor_on
off
# cat kdamond_pid
none
# echo on > monitor_on
# cat kdamond_pid
18594
```

## 使用多个监测线程

每个监测上下文都会创建一个 `kdamond` 线程。你可以使用 `mk_contexts` 和 `rm_contexts` 文件为多个 `kdamond` 需要的用例创建和删除监测上下文。

将新上下文的名称写入 `mk_contexts` 文件，在 DAMON debugfs 目录上创建一个该名称的目录。该目录将有该上下文的 DAMON debugfs 文件：

```
# cd <debugfs>/damon
# ls foo
# ls: cannot access 'foo': No such file or directory
# echo foo > mk_contexts
# ls foo
# attrs init_regions kdamond_pid schemes target_ids
```

如果不再需要上下文，你可以通过把上下文的名字放到 `rm_contexts` 文件中来删除它和相应的目录：

```
# echo foo > rm_contexts
# ls foo
# ls: cannot access 'foo': No such file or directory
```

注意，`mk_contexts`、`rm_contexts` 和 `monitor_on` 文件只在根目录下。

## 监测结果的监测点

DAMON 通过一个 tracepoint `damon:damon_aggregated` 提供监测结果。当监测开启时，你可以记录追踪点事件，并使用追踪点支持工具如 `perf` 显示结果。比如说：

```
# echo on > monitor_on
# perf record -e damon:damon_aggregated &
# sleep 5
# kill 9 $(pidof perf)
```

```
# echo off > monitor_on  
# perf script
```

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/admin-guide/mm/damon/reclaim.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译

## 基于 DAMON 的回收

基于 DAMON 的回收 (DAMON\_RECLAIM) 是一个静态的内核模块，旨在用于轻度内存压力下的主动和轻量级的回收。它的目的不是取代基于 LRU 列表的页面回收，而是有选择地用于不同程度的内存压力和要求。

### 哪些地方需要主动回收？

在一般的内存超量使用 (over-committed systems, 虚拟化相关术语) 的系统上，主动回收冷页有助于节省内存和减少延迟高峰，这些延迟是由直接回收进程或 kswapd 的 CPU 消耗引起的，同时只产生最小的性能下降<sup>12</sup>。

基于空闲页报告<sup>3</sup> 的内存过度承诺的虚拟化系统就是很好的例子。在这样的系统中，客户机向主机报告他们的空闲内存，而主机则将报告的内存重新分配给其他客户。因此，系统的内存得到了充分的利用。然而，客户可能不那么节省内存，主要是因为一些内核子系统和用户空间应用程序被设计为使用尽可能多的内存。然后，客户机可能只向主机报告少量的内存是空闲的，导致系统的内存利用率下降。在客户中运行主动回收可以缓解这个问题。

<sup>1</sup> <https://research.google/pubs/pub48551/>

<sup>2</sup> <https://lwn.net/Articles/787611/>

<sup>3</sup> [https://www.kernel.org/doc/html/latest/vm/free\\_page\\_reporting.html](https://www.kernel.org/doc/html/latest/vm/free_page_reporting.html)

## 它是如何工作的？

DAMON\_RECLAIM 找到在特定时间内没有被访问的内存区域并分页。为了避免它在分页操作中消耗过多的 CPU，可以配置一个速度限制。在这个速度限制下，它首先分页出那些没有被访问过的内存区域。系统管理员还可以配置在什么情况下这个方案应该自动激活和停用三个内存压力水位。

## 接口：模块参数

要使用这个功能，你首先要确保你的系统运行在一个以 CONFIG\_DAMON\_RECLAIM=y 构建的内核上。

为了让系统管理员启用或禁用它，并为给定的系统进行调整，DAMON\_RECLAIM 利用了模块参数。也就是说，你可以把 `damon_reclaim.<parameter>=<value>` 放在内核启动命令行上，或者把适当的值写入 `/sys/modules/damon_reclaim/parameters/<parameter>` 文件。

注意，除启用外的参数值只在 DAMON\_RECLAIM 启动时应用。因此，如果你想在运行时应用新的参数值，而 DAMON\_RECLAIM 已经被启用，你应该通过启用的参数文件禁用和重新启用它。在重新启用之前，应将新的参数值写入适当的参数值中。

下面是每个参数的描述。

### **enabled**

启用或禁用 DAMON\_RECLAIM。

你可以通过把这个参数的值设置为 Y 来启用 DAMON\_RCLAIM，把它设置为 N 可以禁用 DAMON\_RECLAIM。注意，由于基于水位的激活条件，DAMON\_RECLAIM 不能进行真正的监测和回收。这一点请参考下面关于水位参数的描述。

### **min\_age**

识别冷内存区域的时间阈值，单位是微秒。

如果一个内存区域在这个时间或更长的时间内没有被访问，DAMON\_RECLAIM 会将该区域识别为冷的，并回收它。

默认为 120 秒。

### **quota\_ms**

回收的时间限制，以毫秒为单位。

DAMON\_RECLAIM 试图在一个时间窗口 (quota\_reset\_interval\_ms) 内只使用到这个时间，以尝试回收冷页。这可以用来限制 DAMON\_RECLAIM 的 CPU 消耗。如果该值为零，则该限制被禁用。

默认为 10ms。

### **quota\_sz**

回收的内存大小限制，单位为字节。

DAMON\_RECLAIM 收取在一个时间窗口 (quota\_reset\_interval\_ms) 内试图回收的内存量，并使其不超过这个限制。这可以用来限制 CPU 和 IO 的消耗。如果该值为零，则限制被禁用。

默认情况下是 128 MiB。

### **quota\_reset\_interval\_ms**

时间/大小配额收取重置间隔，单位为毫秒。

时间 (quota\_ms) 和大小 (quota\_sz) 的配额的目标重置间隔。也就是说，DAMON\_RECLAIM 在尝试回收 ‘不’ 超过 quota\_ms 毫秒或 quota\_sz 字节的内存。

默认为 1 秒。

### **wmarks\_interval**

当 DAMON\_RECLAIM 被启用但由于其水位规则而不活跃时，在检查水位之前的最小等待时间。

### **wmarks\_high**

高水位的可用内存率（每千字节）。

如果系统的可用内存（以每千字节为单位）高于这个数值，DAMON\_RECLAIM 就会变得不活跃，所以它什么也不做，只是定期检查水位。

## wmarks\_mid

中间水位的可用内存率（每千字节）。

如果系统的空闲内存（以每千字节为单位）在这个和低水位线之间，DAMON\_RECLAIM 就会被激活，因此开始监测和回收。

## wmarks\_low

低水位的可用内存率（每千字节）。

如果系统的空闲内存（以每千字节为单位）低于这个数值，DAMON\_RECLAIM 就会变得不活跃，所以它除了定期检查水位外什么都不做。在这种情况下，系统会退回到基于 LRU 列表的页面粒度回收逻辑。

## sample\_interval

监测的采样间隔，单位是微秒。

DAMON 用于监测冷内存的采样间隔。更多细节请参考 DAMON 文档 ([详细用法](#))。

## aggr\_interval

监测的聚集间隔，单位是微秒。

DAMON 对冷内存监测的聚集间隔。更多细节请参考 DAMON 文档 ([详细用法](#))。

## min\_nr\_regions

监测区域的最小数量。

DAMON 用于冷内存监测的最小监测区域数。这可以用来设置监测质量的下限。但是，设置的太高可能会导致监测开销的增加。更多细节请参考 DAMON 文档 ([详细用法](#))。

## max\_nr\_regions

监测区域的最大数量。

DAMON 用于冷内存监测的最大监测区域数。这可以用来设置监测开销的上限值。但是，设置得太低可能会导致监测质量不好。更多细节请参考 DAMON 文档 ([详细用法](#))。

### **monitor\_region\_start**

目标内存区域的物理地址起点。

DAMON\_RECLAIM 将对其进行工作的内存区域的起始物理地址。也就是说，DAMON\_RECLAIM 将在这个区域中找到冷的内存区域并进行回收。默认情况下，该区域使用最大系统内存区。

### **monitor\_region\_end**

目标内存区域的结束物理地址。

DAMON\_RECLAIM 将对其进行工作的内存区域的末端物理地址。也就是说，DAMON\_RECLAIM 将在这个区域内找到冷的内存区域并进行回收。默认情况下，该区域使用最大系统内存区。

### **kdamond\_pid**

DAMON 线程的 PID。

如果 DAMON\_RECLAIM 被启用，这将成为工作线程的 PID。否则，为-1。

### **nr\_reclaim\_tried\_regions**

试图通过 DAMON\_RECLAIM 回收的内存区域的数量。

### **bytes\_reclaim\_tried\_regions**

试图通过 DAMON\_RECLAIM 回收的内存区域的总字节数。

### **nr\_reclaimed\_regions**

通过 DAMON\_RECLAIM 成功回收的内存区域的数量。

### **bytes\_reclaimed\_regions**

通过 DAMON\_RECLAIM 成功回收的内存区域的总字节数。

## nr\_quota\_exceeds

超过时间/空间配额限制的次数。

### 例子

下面的运行示例命令使 DAMON\_RECLAIM 找到 30 秒或更长时间没有访问的内存区域并“回收”？为了避免 DAMON\_RECLAIM 在分页操作中消耗过多的 CPU 时间，回收被限制在每秒 1GiB 以内。它还要求 DAMON\_RECLAIM 在系统的可用内存率超过 50% 时不做任何事情，但如果它低于 40% 时就开始真正的工作。如果 DAMON\_RECLAIM 没有取得进展，因此空闲内存率低于 20%，它会要求 DAMON\_RECLAIM 再次什么都不做，这样我们就可以退回到基于 LRU 列表的页面粒度回收了：

```
# cd /sys/modules/damon_reclaim/parameters
# echo 30000000 > min_age
# echo $((1 * 1024 * 1024 * 1024)) > quota_sz
# echo 1000 > quota_reset_interval_ms
# echo 500 > wmarks_high
# echo 400 > wmarks_mid
# echo 200 > wmarks_low
# echo Y > enabled
```

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/admin-guide/mm/ksm.rst

翻译 徐鑫 xu xin <[xu.xin16@zte.com.cn](mailto:xu.xin16@zte.com.cn)>

## 内核同页合并

### 概述

KSM 是一种能节省内存的数据去重功能，由 CONFIG\_KSM=y 启用，并在 2.6.32 版本时被添加到 Linux 内核。详见 mm/ksm.c 的实现，以及 <http://lwn.net/Articles/306704> 和 <https://lwn.net/Articles/330589> KSM 最初目的是为了与 KVM（即著名的内核共享内存）一起使用而开发的，通过共享虚拟机之间的公共数据，将更多虚拟机放入物理内存。但它对于任何会生成多个相同数据实例的应用程序都是很有用的。

KSM 的守护进程 `ksmd` 会定期扫描那些已注册的用户内存区域，查找内容相同的页面，这些页面可以被单个写保护页面替换（如果进程以后想要更新其内容，将自动复制）。使用：引用：`sysfs interface <ksm_sysfs>` 接口来配置 KSM 守护程序在单个过程中所扫描的页数以及两个过程之间的间隔时间。

KSM 只合并匿名（私有）页面，从不合并页缓存（文件）页面。KSM 的合并页面最初只能被锁定在内核内存中，但现在可以就像其他用户页面一样被换出（但当它们被交换回来时共享会被破坏：`ksmd` 必须重新发现它们的身份并再次合并）。

### 以 `madvise` 控制 KSM

KSM 仅在特定的地址空间区域时运行，即应用程序通过使用如下所示的 `madvise(2)` 系统调用来请求某块地址成为可能的合并候选者的地址空间：

```
int madvise(addr, length, MADV_MERGEABLE)
```

应用程序当然也可以通过调用：

```
int madvise(addr, length, MADV_UNMERGEABLE)
```

来取消该请求，并恢复为非共享页面：此时 KSM 将去除合并在该范围内的任何合并页。注意：这个去除合并的调用可能突然需要的内存量超过实际可用的内存量-那么可能会出现 `EAGAIN` 失败，但更可能会唤醒 `OOM killer`。

如果 KSM 未被配置到正在运行的内核中，则 `madvise MADV_MERGEABLE` 和 `MADV_UNMERGEABLE` 的调用只会以 `EINVAL` 失败。如果正在运行的内核是用 `CONFIG_KSM=y` 方式构建的，那么这些调用通常会成功：即使 KSM 守护程序当前没有运行，`MADV_MERGEABLE` 仍然会在 KSM 守护程序启动时注册范围，即使该范围不能包含 KSM 实际可以合并的任何页面，即使 `MADV_UNMERGEABLE` 应用于从未标记为 `MADV_MERGEABLE` 的范围。

如果一块内存区域必须被拆分为至少一个新的 `MADV_MERGEABLE` 区域或 `MADV_UNMERGEABLE` 区域，当该进程将超过 `vm.max_map_count` 的设定，则 `madvise` 可能返回 `ENOMEM`。（请参阅文档 `Documentation/admin-guide/sysctl/vm.rst`）。

与其他 `madvise` 调用一样，它们在用户地址空间的映射区域上使用：如果指定的范围包含未映射的间隙（尽管在中间的映射区域工作），它们将报告 `ENOMEM`，如果没有足够的内存用于内部结构，则可能会因 `EAGAIN` 而失败。

### KSM 守护进程 sysfs 接口

KSM 守护进程可以由``/sys/kernel/mm/ksm`` 中的 sysfs 文件控制，所有人都可以读取，但只能由 root 用户写入。各接口解释如下：

**pages\_to\_scan** `ksmd` 进程进入睡眠前要扫描的页数。例如，`echo 100 > /sys/kernel/mm/ksm/pages_to_scan`

默认值：100（该值被选择用于演示目的）

**sleep\_millisecs** ksmd 在下次扫描前应休眠多少毫秒。例如，`echo 20 > /sys/kernel/mm/ksm/sleep_millisecs`

默认值：20（该值被选择用于演示目的）

**merge\_across\_nodes** 指定是否可以合并来自不同 NUMA 节点的页面。当设置为 0 时，ksm 仅合并在物理上位于同一 NUMA 节点的内存区域中的页面。这降低了访问共享页面的延迟。在有明显的 NUMA 距离上，具有更多节点的系统可能受益于设置该值为 0 时的更低延迟。而对于需要对内存使用量最小化的较小系统来说，设置该值为 1（默认设置）则可能会受益于更大共享页面。在决定使用哪种设置之前，您可能希望比较系统在每种设置下的性能。`merge_across_nodes` 仅当系统中没有 ksm 共享页面时，才能被更改设置：首先将接口`run` 设置为 2 从而对页进行去合并，然后在修改 `merge_across_nodes` 后再将‘run’又设置为 1，以根据新设置来重新合并。

默认值：1（如早期的发布版本一样合并跨站点）

## run

- 设置为 0 可停止 ksmd 运行，但保留合并页面，
- 设置为 1 可运行 ksmd，例如，`echo 1 > /sys/kernel/mm/ksm/run`，
- 设置为 2 可停止 ksmd 运行，并且对所有目前已合并的页进行去合并，但保留可合并区域以供下次运行。

默认值：0（必须设置为 1 才能激活 KSM，除非禁用了 CONFIG\_SYSFS）

**use\_zero\_pages** 指定是否应当特殊处理空页（即那些仅含 zero 的已分配页）。当该值设置为 1 时，空页与内核零页合并，而不是像通常情况下那样空页自身彼此合并。这可以根据工作负载的不同，在具有着色零页的架构上可以提高性能。启用此设置时应小心，因为它可能会降低某些工作负载的 KSM 性能，比如，当待合并的候选页面的校验和与空页面的校验和恰好匹配的时候。此设置可随时更改，仅对那些更改后再合并的页面有效。

默认值：0（如同早期版本的 KSM 正常表现）

**max\_page\_sharing** 单个 KSM 页面允许的最大共享站点数。这将强制执行重复数据消除限制，以避免涉及遍历共享 KSM 页面的虚拟映射的虚拟内存操作的高延迟。最小值为 2，因为新创建的 KSM 页面将至少有两个共享者。该值越高，KSM 合并内存的速度越快，去重因子也越高，但是对于任何给定的 KSM 页面，虚拟映射的最坏情况遍历的速度也会越慢。减慢了这种遍历速度就意味着在交换、压缩、NUMA 平衡和页面迁移期间，某些虚拟内存操作将有更高的延迟，从而降低这些虚拟内存操作调用者的响应能力。其他任务如果不涉及执行虚拟映射遍历的 VM 操作，其任务调度延迟不受此参数的影响，因为这些遍历本身是调度友好的。

**stable\_node\_chains\_prune\_millisecs** 指定 KSM 检查特定页面的元数据的频率（即那些达到过时信息数据去重限制标准的页面）单位是毫秒。较小的毫秒值将以更低的延迟来释放 KSM 元数据，但它们将使 ksmd 在扫描期间使用更多 CPU。如果还没有一个 KSM 页面达到 `max_page_sharing` 标准，那就没有什么用。

KSM 与 MADV\_MERGEABLE 的工作有效性体现于 `/sys/kernel/mm/ksm/` 路径下的接口：

**pages\_shared** 表示多少共享页正在被使用

**pages\_sharing** 表示还有多少站点正在共享这些共享页，即节省了多少

**pages\_unshared** 表示有多少页是唯一的，但被反复检查以进行合并

**pages\_volatile** 表示有多少页因变化太快而无法放在 tree 中

**full\_scans** 表示所有可合并区域已扫描多少次

**stable\_node\_chains** 达到 max\_page\_sharing 限制的 KSM 页数

**stable\_node\_dups** 重复的 KSM 页数

比值 pages\_sharing/pages\_shared 的最大值受限制于 max\_page\_sharing 的设定。要想增加该比值，则相应地要增加 max\_page\_sharing 的值。

Todolist: \* concepts \* cma\_debugfs \* hugetlbpage \* idle\_page\_tracking \* memory-hotplug \* nommu-mmap \* numa\_memory\_policy \* numaperf \* pagemap \* soft-dirty \* swap\_numa \* transhuge \* userfaultfd \* zswap

Todolist:

- acpi/index
- aoe/index
- auxdisplay/index
- bcache
- binderfs
- binfmt-misc
- blockdev/index
- bootconfig
- braille-console
- btmrvl
- cgroup-v1/index
- cgroup-v2
- cifs/index
- dell\_rbu
- device-mapper/index
- edid
- efi-stub
- ext4
- nfs/index

- gpio/index
- highuid
- hw\_random
- initrd
- iostats
- java
- jfs
- kernel-per-CPU-kthreads
- laptops/index
- lcd-panel-cgram
- ldm
- LSM/index
- md
- media/index
- module-signing
- mono
- namespaces/index
- numastat
- parport
- perf-security
- pm/index
- pnp
- rapidio
- ras
- rtc
- serial-console
- svga
- thunderbolt
- ufs
- vga-softcursor

- video-output
- xfs

TODOList:

- kbuild/index

### \* 固件相关文档

下列文档描述了内核需要的平台固件相关信息。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

---

**Original** Documentation/Devicetree/index.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译

### \* Open Firmware 和 Devicetree

该文档是整个设备树文档的总目录，标题中多是业内默认的术语，初见就翻译成中文，晦涩难懂，因此尽量保留，后面翻译其子文档时，可能会根据语境，灵活地翻译为中文。

### 内核 Devicetree 的使用

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

---

**Original** Documentation/Devicetree/usage-model.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译

## Linux 和 Devicetree

Linux 对设备树数据的使用模型

作者 Grant Likely <[grant.likely@secretlab.ca](mailto:grant.likely@secretlab.ca)>

这篇文章描述了 Linux 如何使用设备树。关于设备树数据格式的概述可以在 [devicetree.org<sup>1</sup>](https://www.devicetree.org/specifications/) 的设备树使用页面上找到。

“Open Firmware Device Tree”，或简称为 Devicetree (DT)，是一种用于描述硬件的数据结构和语言。更确切地说，它是一种操作系统可读的硬件描述，这样操作系统就不需要对机器的细节进行硬编码。

从结构上看，DT 是一棵树，或者说是带有命名节点的无环图，节点可以有任意数量的命名属性来封装任意的数据。还存在一种机制，可以在自然的树状结构之外创建从一个节点到另一个节点的任意链接。

从概念上讲，一套通用的使用惯例，称为“bindings”（后文译为绑定），被定义为数据应该如何出现在树中，以描述典型的硬件特性，包括数据总线、中断线、GPIO 连接和外围设备。

尽可能使用现有的绑定来描述硬件，以最大限度地利用现有的支持代码，但由于属性和节点名称是简单的文本字符串，通过定义新的节点和属性来扩展现有的绑定或创建新的绑定很容易。然而，要警惕的是，在创建一个新的绑定之前，最好先对已经存在的东西做一些功课。目前有两种不同的、不兼容的 i2c 总线的绑定，这是因为创建新的绑定时没有事先调查 i2c 设备在现有系统中是如何被枚举的。

### 1. 历史

DT 最初是由 Open Firmware 创建的，作为将数据从 Open Firmware 传递给客户程序（如传递给操作系统）的通信方法的一部分。操作系统使用设备树在运行时探测硬件的拓扑结构，从而在没有硬编码信息的情况下支持大多数可用的硬件（假设所有设备的驱动程序都可用）。

由于 Open Firmware 通常在 PowerPC 和 SPARC 平台上使用，长期以来，对这些架构的 Linux 支持一直使用设备树。

2005 年，当 PowerPC Linux 开始大规模清理并合并 32 位和 64 位支持时，决定在所有 Powerpc 平台上要求 DT 支持，无论它们是否使用 Open Firmware。为了做到这一点，我们创建了一个叫做扁平化设备树 (FDT) 的 DT 表示法，它可以作为一个二进制的 blob 传递给内核，而不需要真正的 Open Firmware 实现。U-Boot、kexec 和其他引导程序被修改，以支持传递设备树二进制 (dtb) 和在引导时修改 dtb。DT 也被添加到 PowerPC 引导包装器 (arch/powerpc/boot/\*) 中，这样 dtb 就可以被包裹在内核镜像中，以支持引导现有的非 DT 察觉的固件。

一段时间后，FDT 基础架构被普及到了所有的架构中。在写这篇文章的时候，6 个主线架构 (arm、microblaze、mips、powerpc、sparc 和 x86) 和 1 个非主线架构 (ios) 有某种程度的 DT 支持。

<sup>1</sup> <https://www.devicetree.org/specifications/>

## 1. 数据模型

如果你还没有读过设备树用法 [1]\_ 页，那么现在就去读吧。没关系，我等着…。

### 2.1 高层次视角

最重要的是要明白，DT 只是一个描述硬件的数据结构。它没有什么神奇之处，也不会神奇地让所有的硬件配置问题消失。它所做的是提供一种语言，将硬件配置与 Linux 内核（或任何其他操作系统）中的板卡和设备驱动支持解耦。使用它可以使板卡和设备支持变成数据驱动；根据传递到内核的数据做出设置决定，而不是根据每台机器的硬编码选择。

理想情况下，数据驱动的平台设置应该导致更少的代码重复，并使其更容易用一个内核镜像支持各种硬件。

Linux 使用 DT 数据有三个主要目的：

- 1) 平台识别。
- 2) 运行时配置，以及
- 3) 设备数量。

### 2.2 平台识别

首先，内核将使用 DT 中的数据来识别特定的机器。在一个理想的世界里，具体的平台对内核来说并不重要，因为所有的平台细节都会被设备树以一致和可靠的方式完美描述。但是，硬件并不完美，所以内核必须在早期启动时识别机器，以便有机会运行特定于机器的修复程序。

在大多数情况下，机器的身份是不相关的，而内核将根据机器的核心 CPU 或 SoC 来选择设置代码。例如，在 ARM 上，arch/arm/kernel/setup.c 中的 setup\_arch() 将调用 arch/arm/kernel/devtree.c 中的 setup\_machine\_fdt()，它通过 machine\_desc 表搜索并选择与设备树数据最匹配的 machine\_desc。它通过查看根设备树节点中的' compatible' 属性，并将其与 struct machine\_desc 中的 dt\_compatible 列表（如果你好奇，该列表定义在 arch/arm/include/asm/mach/arch.h 中）进行比较，从而确定最佳匹配。

“compatible”属性包含一个排序的字符串列表，以机器的确切名称开始，后面是一个可选的与之兼容的板子列表，从最兼容到最不兼容排序。例如，TI BeagleBoard 和它的后继者 BeagleBoard xM 板的根兼容属性可能看起来分别为：

```
compatible = "ti,omap3-beagleboard", "ti,omap3450", "ti,omap3";
compatible = "ti,omap3-beagleboard-xm", "ti,omap3450", "ti,omap3";
```

其中“ti,omap3-beagleboard-xm”指定了确切的型号，它还声称它与 OMAP 3450 SoC 以及一般的 OMP3 系列 SoC 兼容。你会注意到，该列表从最具体的（确切的板子）到最不具体的（SoC 系列）进行排序。

聪明的读者可能会指出，Beagle xM 也可以声称与原 Beagle 板兼容。然而，我们应该当心在板级上这样做，因为通常情况下，即使在同一产品系列中，每块板都有很高的变化，而且当一块板声称与另一块板兼容时，很难确定到底是什么意思。对于高层来说，最好是谨慎行事，不要声称一块板子与另一块板子兼容。值得注意的例外是，当一块板子是另一块板子的载体时，例如 CPU 模块连接到一个载体板上。

关于兼容值还有一个注意事项。在兼容属性中使用的任何字符串都必须有文件说明它表示什么。在 Documentation/devicetree/bindings 中添加兼容字符串的文档。

同样在 ARM 上，对于每个 machine\_desc，内核会查看是否有任何 dt\_compatible 列表条目出现在兼容属性中。如果有，那么该 machine\_desc 就是驱动该机器的候选者。在搜索了整个 machine\_descriptions 表之后，setup\_machine\_fdt() 根据每个 machine\_desc 在兼容属性中匹配的条目，返回“最兼容”的 machine\_desc。如果没有找到匹配的 machine\_desc，那么它将返回 NULL。

这个方案背后的原因是观察到，在大多数情况下，如果它们都使用相同的 SoC 或相同系列的 SoC，一个 machine\_desc 可以支持大量的电路板。然而，不可避免地会有一些例外情况，即特定的板子需要特殊的设置代码，这在一般情况下是没有用的。特殊情况可以通过在通用设置代码中明确检查有问题的板子来处理，但如果超过几个情况下，这样做很快就会变得很难看和/或无法维护。

相反，兼容列表允许通用 machine\_desc 通过在 dt\_compatible 列表中指定“不太兼容”的值来提供对广泛的通用板的支持。在上面的例子中，通用板支持可以声称与“ti,ompa3”或“ti,ompa3450”兼容。如果在最初的 beagleboard 上发现了一个 bug，需要在早期启动时使用特殊的变通代码，那么可以添加一个新的 machine\_desc，实现变通，并且只在“ti,omap3-beagleboard”上匹配。

PowerPC 使用了一个稍微不同的方案，它从每个 machine\_desc 中调用 .probe() 钩子，并使用第一个返回 TRUE 的钩子。然而，这种方法没有考虑到兼容列表的优先级，对于新的架构支持可能应该避免。

## 2.3 运行时配置

在大多数情况下，DT 是将数据从固件传递给内核的唯一方法，所以也被用来传递运行时和配置数据，如内核参数字符串和 initrd 镜像的位置。

这些数据大部分都包含在 chosen 节点中，当启动 Linux 时，它看起来就像这样：

```
chosen {
    bootargs = "console=ttyS0,115200 loglevel=8";
    initrd-start = <0xc800000>;
    initrd-end = <0xc8200000>;
};
```

bootargs 属性包含内核参数，initrd-\* 属性定义 initrd blob 的地址和大小。注意 initrd-end 是 initrd 映像后的第一个地址，所以这与结构体资源的通常语义不一致。选择的节点也可以选择包含任意数量的额外属性，用于平台特定的配置数据。

在早期启动过程中，架构设置代码通过不同的辅助回调函数多次调用 of\_scan\_flat\_dt() 来解析设备树数据，然后进行分页设置。of\_scan\_flat\_dt() 代码扫描设备树，并使用辅助函数来提取早期启动期间所需的信息。通常情况下，early\_init\_dt\_scan\_chosen() 辅助函数用于解析所选节点，包括内核参数，early\_init\_dt\_scan\_root() 用于初始化 DT 地址空间模型，early\_init\_dt\_scan\_memory() 用于确定可用 RAM 的大小和位置。

在 ARM 上，函数 setup\_machine\_fdt() 负责在选择支持板子的正确 machine\_desc 后，对设备树进行早期扫描。

## 2.4 设备数量

在电路板被识别后，在早期配置数据被解析后，内核初始化可以以正常方式进行。在这个过程中的某个时刻，`unflatten_device_tree()` 被调用以将数据转换成更有效的运行时表示。这也是调用机器特定设置钩子的时候，比如 ARM 上的 `machine_desc .init_early()`、`.init_irq()` 和 `.init_machine()` 钩子。本节的其余部分使用了 ARM 实现的例子，但所有架构在使用 DT 时都会做几乎相同的事情。

从名称上可以猜到，`.init_early()` 用于在启动过程早期需要执行的任何机器特定设置，而`.init_irq()` 则用于设置中断处理。使用 DT 并不会实质性地改变这两个函数的行为。如果提供了 DT，那么`.init_early()` 和`.init_irq()` 都能调用任何一个 DT 查询函数（`of_*` in `include/linux/of*.h`），以获得关于平台的额外数据。

DT 上下文中最有趣的钩子是`.init_machine()`，它主要负责将平台的数据填充到 Linux 设备模型中。历史上，这在嵌入式平台上是通过在板卡 `support.c` 文件中定义一组静态时钟结构、`platform_devices` 和其他数据，并在`.init_machine()` 中大量注册来实现的。当使用 DT 时，就不用为每个平台的静态设备进行硬编码，可以通过解析 DT 获得设备列表，并动态分配设备结构体。

最简单的情况是，`.init_machine()` 只负责注册一个 `platform_devices`。`platform_device` 是 Linux 使用的一个概念，用于不能被硬件检测到的内存或 I/O 映射的设备，以及“复合”或“虚拟”设备（后面会详细介绍）。虽然 DT 没有“平台设备”的术语，但平台设备大致对应于树根的设备节点和简单内存映射总线节点的子节点。

现在是举例说明的好时机。下面是 NVIDIA Tegra 板的设备树的一部分：

```
/{
    compatible = "nvidia,harmony", "nvidia,tegra20";
    #address-cells = <1>;
    #size-cells = <1>;
    interrupt-parent = <&intc>;

    chosen { };
    aliases { };

    memory {
        device_type = "memory";
        reg = <0x00000000 0x40000000>;
    };

    soc {
        compatible = "nvidia,tegra20-soc", "simple-bus";
        #address-cells = <1>;
        #size-cells = <1>;
        ranges;

        intc: interrupt-controller@50041000 {
            compatible = "nvidia,tegra20-gic";
            interrupt-controller;
            #interrupt-cells = <1>;
            reg = <0x50041000 0x1000>, < 0x50040100 0x0100 >;
        };
    };
}
```

```

serial@70006300 {
    compatible = "nvidia,tegra20-uart";
    reg = <0x70006300 0x100>;
    interrupts = <122>;
};

i2s1: i2s@70002800 {
    compatible = "nvidia,tegra20-i2s";
    reg = <0x70002800 0x100>;
    interrupts = <77>;
    codec = <&wm8903>;
};

i2c@7000c000 {
    compatible = "nvidia,tegra20-i2c";
    #address-cells = <1>;
    #size-cells = <0>;
    reg = <0x7000c000 0x100>;
    interrupts = <70>;

    wm8903: codec@1a {
        compatible = "wlf,wm8903";
        reg = <0x1a>;
        interrupts = <347>;
    };
};
};

sound {
    compatible = "nvidia,harmony-sound";
    i2s-controller = <&i2s1>;
    i2s-codec = <&wm8903>;
};
};

```

在.init\_machine() 时，Tegra 板支持代码将需要查看这个 DT，并决定为哪些节点创建 platform\_devices。然而，看一下这个树，并不能立即看出每个节点代表什么类型的设备，甚至不能看出一个节点是否代表一个设备。/chosen、/aliases 和 /memory 节点是信息节点，并不描述设备（尽管可以说内存可以被认为是一个设备）。/soc 节点的子节点是内存映射的设备，但是 codec@1a 是一个 i2c 设备，而 sound 节点代表的不是一个设备，而是其他设备是如何连接在一起以创建音频子系统的。我知道每个设备是什么，因为我熟悉电路板的设计，但是内核怎么知道每个节点该怎么做？

诀窍在于，内核从树的根部开始，寻找具有“兼容”属性的节点。首先，一般认为任何具有“兼容”属性的节点都代表某种设备；其次，可以认为树根的任何节点要么直接连接到处理器总线上，要么是无法用其他方式描述的杂项系统设备。对于这些节点中的每一个，Linux 都会分配和注册一个 platform\_device，它又可能被绑定到一个 platform\_driver。

为什么为这些节点使用 platform\_device 是一个安全的假设？嗯，就 Linux 对设备的建模方式而言，几乎

所有的总线类型都假定其设备是总线控制器的孩子。例如，每个 `i2c_client` 是 `i2c_master` 的一个子节点。每个 `spi_device` 都是 SPI 总线的一个子节点。类似的还有 USB、PCI、MDIO 等。同样的层次结构也出现在 DT 中，I2C 设备节点只作为 I2C 总线节点的子节点出现。同理，SPI、MDIO、USB 等等。唯一不需要特定类型的父设备的设备是 `platform_devices`（和 `amba_devices`，但后面会详细介绍），它们将愉快地运行在 Linux/sys/devices 树的底部。因此，如果一个 DT 节点位于树的根部，那么它真的可能最好注册为 `platform_device`。

Linux 板支持代码调用 `of_platform_populate(NULL, NULL, NULL, NULL)` 来启动树根的设备发现。参数都是 NULL，因为当从树的根部开始时，不需要提供一个起始节点（第一个 NULL），一个父结构设备（最后一个 NULL），而且我们没有使用匹配表（尚未）。对于只需要注册设备的板子，除了 `of_platform_populate()` 的调用，`.init_machine()` 可以完全为空。

在 Tegra 的例子中，这说明了 `/soc` 和 `/sound` 节点，但是 SoC 节点的子节点呢？它们不应该也被注册为平台设备吗？对于 Linux DT 支持，一般的行为是子设备在驱动 `.probe()` 时被父设备驱动注册。因此，一个 I2C 总线设备驱动程序将为每个子节点注册一个 `i2c_client`，一个 SPI 总线驱动程序将注册其 `spi_device` 子节点，其他总线类型也是如此。根据该模型，可以编写一个与 SoC 节点绑定的驱动程序，并简单地为其每个子节点注册 `platform_device`。板卡支持代码将分配和注册一个 SoC 设备，一个（理论上的）SoC 设备驱动程序可以绑定到 SoC 设备，并在其 `.probe()` 钩中为 `/soc/interruptcontroller`、`/soc/serial`、`/soc/i2s` 和 `/soc/i2c` 注册 `platform_devices`。很简单，对吗？

实际上，事实证明，将一些 `platform_device` 的子设备注册为更多的 `platform_device` 是一种常见的模式，设备树支持代码反映了这一点，并使上述例子更简单。`of_platform_populate()` 的第二个参数是一个 `of_device_id` 表，任何与该表中的条目相匹配的节点也将获得其子节点的注册。在 Tegra 的例子中，代码可以是这样的：

```
static void __init harmony_init_machine(void)
{
    /* ... */
    of_platform_populate(NULL, of_default_bus_match_table, NULL, NULL);
}
```

“simple-bus” 在 Devicetree 规范中被定义为一个属性，意味着一个简单的内存映射的总线，所以 `of_platform_populate()` 代码可以被写成只是假设简单总线兼容的节点将总是被遍历。然而，我们把它作为一个参数传入，以便电路板支持代码可以随时覆盖默认行为。

[需要添加关于添加 i2c/spi/etc 子设备的讨论]。

## 附录 A：AMBA 设备

ARM Primecell 是连接到 ARM AMBA 总线的某种设备，它包括对硬件检测和电源管理的一些支持。在 Linux 中，`amba_device` 和 `amba_bus_type` 结构体被用来表示 Primecell 设备。然而，棘手的一点是，AMBA 总线上的所有设备并非都是 Primecell，而且对于 Linux 来说，典型的情况是 `amba_device` 和 `platform_device` 实例都是同一总线段的同义词。

当使用 DT 时，这给 `of_platform_populate()` 带来了问题，因为它必须决定是否将每个节点注册为 `platform_device` 或 `amba_device`。不幸的是，这使设备创建模型变得有点复杂，但解决方案原来并不是太具

有侵略性。如果一个节点与“arm,amba-primecell”兼容，那么 `of_platform_populate()` 将把它注册为 `amba_device` 而不是 `platform_device`。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

**Original** Documentation/Devicetree/of\_unittest.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译

## Open Firmware Devicetree 单元测试

作者: Gaurav Minocha <[gaurav.minocha.os@gmail.com](mailto:gaurav.minocha.os@gmail.com)>

### 1. 概述

本文档解释了执行 OF 单元测试所需的测试数据是如何动态地附加到实时树上的，与机器的架构无关。

建议在继续读下去之前，先阅读以下文件。

- (1) Documentation/devicetree/usage-model.rst
- (2) [http://www.devicetree.org/Device\\_Tree\\_Usage](http://www.devicetree.org/Device_Tree_Usage)

OF Selftest 被设计用来测试提供给设备驱动开发者的接口 (`include/linux/of.h`)，以从未扁平化的设备树数据结构中获取设备信息等。这个接口被大多数设备驱动在各种使用情况下使用。

### 2. 测试数据

设备树源文件 (`drivers/of/unittest-data/testcases.dts`) 包含执行 `drivers/of/unittest.c` 中自动化单元测试所需的测试数据。目前，以下设备树源包含文件 (.dtsi) 被包含在 `testcases.dt` 中：

```
drivers/of/unittest-data/tests-interrupts.dtsi
drivers/of/unittest-data/tests-platform.dtsi
drivers/of/unittest-data/tests-phandle.dtsi
drivers/of/unittest-data/tests-match.dtsi
```

当内核在启用 `OF_SELFTEST` 的情况下被构建时，那么下面的 make 规则：

```
$(obj)/%.dtb: $(src)/%.dts FORCE  
    $(call if_changed_dep, dtc)
```

用于将 DT 源文件 (testcases.dts) 编译成二进制 blob (testcases.dtb)，也被称为扁平化的 DT。

之后，使用以下规则将上述二进制 blob 包装成一个汇编文件（`testcases.dtb.S`）：

```
$(obj)/%.dtb.S: $(obj)/%.dtb  
          $(call cmd, dt S dtb)
```

汇编文件被编译成一个对象文件 (testcases.dtb.o)，并被链接到内核镜像中。

## 2.1. 添加测试数据

未扁平化的设备树结构体：

未扁平化的设备树由连接的设备节点组成，其树状结构形式如下所述：

```
// following struct members are used to construct the tree
struct device_node {
    ...
    struct device_node *parent;
    struct device_node *child;
    struct device_node *sibling;
    ...
};
```

图 1 描述了一个机器的未扁平化设备树的通用结构，只考虑了子节点和同级指针。存在另一个指针，`*parent`，用于反向遍历该树。因此，在一个特定的层次上，子节点和所有的兄弟姐妹节点将有一个指向共同节点的父指针（例如，`child1`、`sibling2`、`sibling3`、`sibling4` 的父指针指向根节点）：

```
graph TD; root["root ('/')"] --- child1["child1 -> sibling2 -> sibling3 -> sibling4 -> null"]; root --- child2["child21 -> sibling22 -> sibling23 -> null"]; root --- child3["child31 -> sibling32 -> null"]; child1 --- sibling2["sibling2"]; child1 --- sibling3["sibling3"]; child1 --- sibling4["sibling4"]; child1 --- null1["null"]; child2 --- sibling21["sibling21"]; child2 --- sibling22["sibling22"]; child2 --- sibling23["sibling23"]; child3 --- sibling31["sibling31"]; child3 --- sibling32["sibling32"];
```



Figure 1: 未扁平化的设备树的通用结构

在执行 OF 单元测试之前，需要将测试数据附加到机器的设备树上（如果存在）。因此，当调用 `self-test_data_add()` 时，首先会读取通过以下内核符号链接到内核镜像中的扁平化设备树数据：

```

__dtb_testcases_begin - address marking the start of test data blob
__dtb_testcases_end   - address marking the end of test data blob

```

其次，它调用 `of_fdt_unflatten_tree()` 来解除扁平化的 blob。最后，如果机器的设备树（即实时树）是存在的，那么它将未扁平化的测试数据树附加到实时树上，否则它将自己作为实时设备树附加。

`attach_node_and_children()` 使用 `of_attach_node()` 将节点附加到实时树上，如下所述。为了解释这一点，图 2 中描述的测试数据树被附加到图 1 中描述的实时树上：

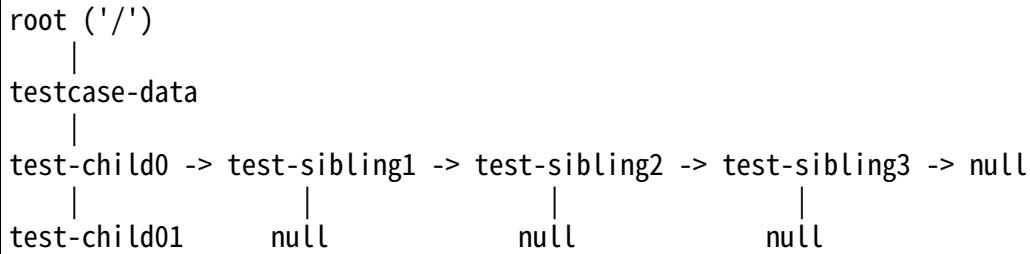
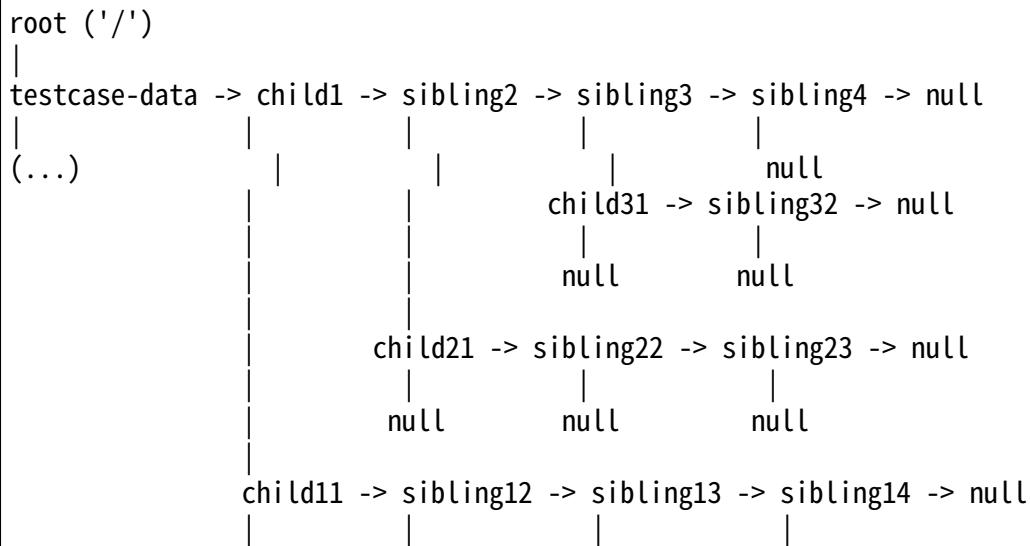


Figure 2: 将测试数据树附在实时树上的例子。

根据上面的方案，实时树已经存在，所以不需要附加根（‘/’）节点。所有其他节点都是通过在每个节点上调用 `of_attach_node()` 来附加的。

在函数 `of_attach_node()` 中，新的节点被附在实时树中给定的父节点的子节点上。但是，如果父节点已经有了一个孩子，那么新节点就会取代当前的孩子，并将其变成其兄弟姐妹。因此，当测试案例的数据节点被连接到上面的实时树（图 1）时，最终的结构如图 3 所示：



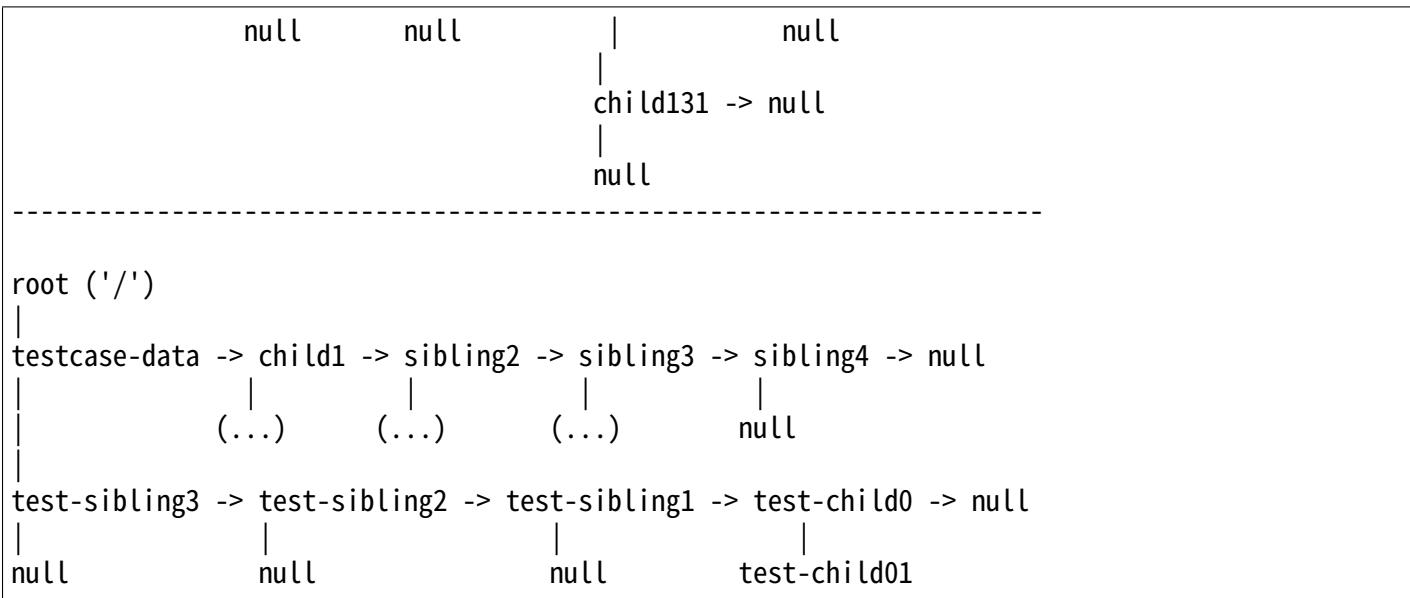


Figure 3: 附加测试案例数据后的实时设备树结构。

聪明的读者会注意到，与先前的结构相比，`test-child0` 节点成为最后一个兄弟姐妹（图 2）。在连接了第一个 `test-child0` 节点之后，又连接了 `test-sibling1` 节点，该节点推动子节点（即 `test-child0`）成为兄弟姐妹，并使自己成为子节点，如上所述。

如果发现一个重复的节点（即如果一个具有相同 `full_name` 属性的节点已经存在于实时树中），那么该节点不会被附加，而是通过调用函数 `update_node_properties()` 将其属性更新到活树的节点中。

## 2.2. 删除测试数据

一旦测试用例执行完，`selftest_data_remove` 被调用，以移除最初连接的设备节点（首先是叶子节点被分离，然后向上移动父节点被移除，最后是整个树）。`selftest_data_remove()` 调用 `detach_node_and_children()`，使用 `of_detach_node()` 将节点从实时设备树上分离。

为了分离一个节点，`of_detach_node()` 要么将给定节点的父节点的子节点指针更新为其同级节点，要么根据情况将前一个同级节点附在给定节点的同级节点上。就这样吧。:)

Todolist:

- kernel-api

## Devicetree Overlays

Todolist:

- changesets
- dynamic-resolution-notes
- overlay-notes

## Devicetree Bindings

Todolist:

- bindings/index

TODOList:

- firmware-guide/index

## \* 应用程序开发人员文档

用户空间 API 手册涵盖了描述应用程序开发人员可见内核接口方面的文档。

TODOlist:

- userspace-api/index

## \* 内核开发简介

这些手册包含有关如何开发内核的整体信息。内核社区非常庞大，一年下来有数千名开发人员做出贡献。与任何大型社区一样，知道如何完成任务将使得更改合并的过程变得更加容易。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/process/index.rst

**Translator** Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

## \* 与 Linux 内核社区一起工作

你想成为 Linux 内核开发人员吗？欢迎之至！在学习许多关于内核的技术知识的同时，了解我们社区的工作方式也很重要。阅读这些文档可以让您以更轻松的、麻烦更少的方式将更改合并到内核。

以下是每位开发人员都应阅读的基本指南：

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

### Original Documentation/process/howto.rst

译者：

英文版维护者： Greg Kroah-Hartman <[greg@kroah.com](mailto:greg@kroah.com)>  
中文版维护者： 李阳 Li Yang <[leoyang.li@nxp.com](mailto:leoyang.li@nxp.com)>  
中文版翻译者： 李阳 Li Yang <[leoyang.li@nxp.com](mailto:leoyang.li@nxp.com)>  
时奎亮 Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>  
中文版校译者：  
    钟宇 TripleX Chung <[xxx.phy@gmail.com](mailto:xxx.phy@gmail.com)>  
    陈琦 Maggie Chen <[chenqi@beyondsoft.com](mailto:chenqi@beyondsoft.com)>  
    王聪 Wang Cong <[xiyou.wangcong@gmail.com](mailto:xiyou.wangcong@gmail.com)>

## 如何参与 Linux 内核开发

这是一篇将如何参与 Linux 内核开发的相关问题一网打尽的终极秘笈。它将指导你成为一名 Linux 内核开发者，并且学会如何同 Linux 内核开发社区合作。它尽可能不包括任何关于内核编程的技术细节，但会给你指引一条获得这些知识的正确途径。

如果这篇文章中的任何内容不再适用，请给文末列出的文件维护者发送补丁。

## 入门

你想了解如何成为一名 Linux 内核开发者？或者老板吩咐你“给这个设备写个 Linux 驱动程序”？这篇文章的目的就是教会你达成这些目标的全部诀窍，它将描述你需要经过的流程以及给出如何同内核社区合作的一些提示。它还将试图解释内核社区为何这样运作。

Linux 内核大部分是由 C 语言写成的，一些体系结构相关的代码用到了汇编语言。要参与内核开发，你必须精通 C 语言。除非你想为某个架构开发底层代码，否则你并不需要了解（任何体系结构的）汇编语言。下面列举的书籍虽然不能替代扎实的 C 语言教育和多年的开发经验，但如果需要的话，做为参考还是不错的：

- “The C Programming Language” by Kernighan and Ritchie [Prentice Hall] 《C 程序设计语言（第 2 版·新版）》（徐宝文李志译）[机械工业出版社]
- “Practical C Programming” by Steve Oualline [O’ Reilly] 《实用 C 语言编程（第三版）》（郭大海译）[中国电力出版社]

- “C: A Reference Manual” by Harbison and Steele [Prentice Hall] 《C 语言参考手册（原书第 5 版）》（邱仲潘等译）[机械工业出版社]

Linux 内核使用 GNU C 和 GNU 工具链开发。虽然它遵循 ISO C89 标准，但也用到了一些标准中没有定义的扩展。内核是自给自足的 C 环境，不依赖于标准 C 库的支持，所以并不支持 C 标准中的部分定义。比如 `long long` 类型的大数除法和浮点运算就不允许使用。有时候确实很难弄清楚内核对工具链的要求和它所使用的扩展，不幸的是目前还没有明确的参考资料可以解释它们。请查阅 `gcc` 信息页（使用“`info gcc`”命令显示）获得一些这方面信息。

请记住你是在学习怎么和已经存在的开发社区打交道。它由一群形形色色的人组成，他们对代码、风格和过程有着很高的标准。这些标准是在长期实践中总结出来的，适应于地理上分散的大型开发团队。它们已经被很好得整理成档，建议你在开发之前尽可能多的学习这些标准，而不要期望别人来适应你或者你公司的行为方式。

## 法律问题

Linux 内核源代码都是在 GPL（通用公共许可证）的保护下发布的。要了解这种许可的细节请查看源代码主目录下的 COPYING 文件。Linux 内核许可准则和如何使用 SPDX <<https://spdx.org/>> 标志符说明在这个文件中 [Documentation/translations/zh\\_CN/process/license-rules.rst](#) 如果你对它还有更深入问题请联系律师，而不要在 Linux 内核邮件组上提问。因为邮件组里的人并不是律师，不要期望他们的话有法律效力。

对于 GPL 的常见问题和解答，请访问以下链接：<https://www.gnu.org/licenses/gpl-faq.html>

## 文档

Linux 内核代码中包含有大量的文档。这些文档对于学习如何与内核社区互动有着不可估量的价值。当一个新的功能被加入内核，最好把解释如何使用这个功能的文档也放进内核。当内核的改动导致面向用户空间的接口发生变化时，最好将相关信息或手册页 (manpages) 的补丁发到 [mtk.manpages@gmail.com](mailto:mtk.manpages@gmail.com)，以向手册页 (manpages) 的维护者解释这些变化。

以下是内核代码中需要阅读的文档：

**Documentation/admin-guide/README.rst** 文件简要介绍了 Linux 内核的背景，并且描述了如何配置和编译内核。内核的新用户应该从这里开始。

**Documentation/process/changes.rst** 文件给出了用来编译和使用内核所需要的最小软件包列表。

**Documentation/translations/zh\_CN/process/coding-style.rst** 描述 Linux 内核的代码风格和理由。所有新代码需要遵守这篇文档中定义的规范。大多数维护者只会接收符合规定的补丁，很多人也只会帮忙检查符合风格的代码。

[Documentation/translations/zh\\_CN/process/submitting-patches.rst](#)  
[Documentation/process/submitting-drivers.rst](#)

这两份文档明确描述如何创建和发送补丁，其中包括（但不仅限于）：

- 邮件内容

- 邮件格式
- 选择收件人

遵守这些规定并不能保证提交成功（因为所有补丁需要通过严格的内容和风格审查），但是忽视他们几乎就意味着失败。

其他关于如何正确地生成补丁的优秀文档包括：“The Perfect Patch”

<https://www.ozlabs.org/~akpm/stuff/tpp.txt>

“Linux kernel patch submission format”

<https://web.archive.org/web/20180829112450/http://linux.yyz.us/patch-format.html>

**Documentation/translations/zh\_CN/process/stable-api-nonsense.rst** 论证内核为什么特意不包括稳定的内核内部 API，也就是说不包括像这样的特性：

- 子系统中间层（为了兼容性？）
- 在不同操作系统间易于移植的驱动程序
- 减缓（甚至阻止）内核代码的快速变化

这篇文档对于理解 Linux 的开发哲学至关重要。对于将开发平台从其他操作系统转移到 Linux 的人来说也很重要。

**Documentation/admin-guide/security-bugs.rst** 如果你认为自己发现了 Linux 内核的安全性问题，请根据这篇文档中的步骤来提醒其他内核开发者并帮助解决这个问题。

*Documentation/translations/zh\_CN/process/management-style.rst*

描述内核维护者的工作方法及其共有特点。这对于刚刚接触内核开发（或者对它感到好奇）的人来说很重要，因为它解释了很多对于内核维护者独特行为的普遍误解与迷惑。

**Documentation/process/stable-kernel-rules.rst** 解释了稳定版内核发布的规则，以及如何将改动放入这些版本的步骤。

**Documentation/process/kernel-docs.rst** 有助于内核开发的外部文档列表。如果你在内核自带的文档中没有找到你想找的内容，可以查看这些文档。

**Documentation/process/applying-patches.rst** 关于补丁是什么以及如何将它打在不同内核开发分支上的好介绍

内核还拥有大量从代码自动生成或者从 ReStructuredText(ReST) 标记生成的文档，比如这个文档，它包含内核内部 API 的全面介绍以及如何妥善处理加锁的规则。所有这些文档都可以通过运行以下命令从内核代码中生成为 PDF 或 HTML 文档：

```
make pdfdocs  
make htmldocs
```

ReST 格式的文档会生成在 Documentation/output. 目录中。它们也可以用下列命令生成 LaTeX 和 ePub 格式文档：

```
make latexdocs  
make epubdocs
```

## 如何成为内核开发者

如果你对 Linux 内核开发一无所知，你应该访问“Linux 内核新手”计划：

<https://kernelnewbies.org>

它拥有一个可以问各种最基本的内核开发问题的邮件列表（在提问之前一定要记得查找已往的邮件，确认是否有人已经回答过相同的问题）。它还拥有一个可以获得实时反馈的 IRC 聊天频道，以及大量对于学习 Linux 内核开发相当有帮助的文档。

网站简要介绍了源代码组织结构、子系统划分以及目前正在进行的项目（包括内核中的和单独维护的）。它还提供了一些基本的帮助信息，比如如何编译内核和打补丁。

如果你想加入内核开发社区并协助完成一些任务，却找不到从哪里开始，可以访问“Linux 内核房管员”计划：

<https://kernelnewbies.org/KernelJanitors>

这是极佳的起点。它提供一个相对简单的任务列表，列出内核代码中需要被重新整理或者改正的地方。通过和负责这个计划的开发者们一同工作，你会学到将补丁集成进内核的基本原理。如果还没有决定下一步要做什么的话，你还可能会得到方向性的指点。

在真正动手修改内核代码之前，理解要修改的代码如何运作是必需的。要达到这个目的，没什么办法比直接读代码更有效了（大多数花招都会有相应的注释），而且一些特制的工具还可以提供帮助。例如，“Linux 代码交叉引用”项目就是一个值得特别推荐的帮助工具，它将源代码显示在有编目和索引的网页上。其中一个更新及时的内核源码库，可以通过以下地址访问：

<https://elixir.bootlin.com/>

## 开发流程

目前 Linux 内核开发流程包括几个“主内核分支”和很多子系统相关的内核分支。这些分支包括：

- Linus 的内核源码树
- 多个主要版本的稳定版内核树
- 子系统相关的内核树
- linux-next 集成测试树

### 主线树

主线树是由 Linus Torvalds 维护的。你可以在 <https://kernel.org> 网站或者代码库中下找到它。它的开发遵循以下步骤：

- 每当一个新版本的内核被发布，为期两周的集成窗口将被打开。在这段时间里维护者可以向 Linus 提交大段的修改，通常这些修改已经被放到-mm 内核中几个星期了。提交大量修改的首选方式是使用 git 工具（内核的代码版本管理工具，更多的信息可以在 <https://git-scm.com/> 获取），不过使用普通补丁也是可以的。
- 两个星期以后-rc1 版本内核发布。之后只有不包含可能影响整个内核稳定性的新功能的补丁才可能被接受。请注意一个全新的驱动程序（或者文件系统）有可能在-rc1 后被接受是因为这样的修改完全独立，不会影响其他的代码，所以没有造成内核退步的风险。在-rc1 以后也可以用 git 向 Linus 提交补丁，不过所有的补丁需要同时被发送到相应的公众邮件列表以征询意见。
- 当 Linus 认为当前的 git 源码树已经达到一个合理健全的状态足以发布供人测试时，一个新的-rc 版本就会被发布。计划是每周都发布新的-rc 版本。
- 这个过程一直持续下去直到内核被认为达到足够稳定的状态，持续时间大概是 6 个星期。

关于内核发布，值得一提的是 **Andrew Morton** 在 **linux-kernel** 邮件列表中如是说：“没有人知道新内核何时会被发布，因为发布是根据已知 bug 的情况来决定的，而不是根据一个事先制定好的时间表。”

### 子系统特定树

各种内核子系统的维护者——以及许多内核子系统开发人员——在源代码库中公开了他们当前的开发状态。这样，其他人就可以看到内核的不同区域发生了什么。在开发速度很快的领域，可能会要求开发人员将提交的内容建立在这样的子系统内核树上，这样就避免了提交与其他已经进行的工作之间的冲突。

这些存储库中的大多数都是 Git 树，但是也有其他的 scm 在使用，或者补丁队列被发布为 Quilt 系列。这些子系统存储库的地址列在 MAINTAINERS 文件中。其中许多可以在 <https://git.kernel.org/> 上浏览。

在将一个建议的补丁提交到这样的子系统树之前，需要对它进行审查，审查主要发生在邮件列表上（请参见下面相应部分）。对于几个内核子系统，这个审查过程是通过工具补丁跟踪的。Patchwork 提供了一个 Web 界面，显示补丁发布、对补丁的任何评论或修订，维护人员可以将补丁标记为正在审查、接受或拒绝。大多数补丁网站都列在 <https://patchwork.kernel.org/>

### Linux-next 集成测试树

在将子系统树的更新合并到主线树之前，需要对它们进行集成测试。为此，存在一个特殊的测试存储库，其中几乎每天都会提取所有子系统树：

<https://git.kernel.org/?p=linux/kernel/git/next/linux-next.git>

通过这种方式，Linux-next 对下一个合并阶段将进入主线内核的内容给出了一个概要展望。非常欢冒险的测试者运行测试 Linux-next。

## 多个主要版本的稳定版内核树

由 3 个数字组成的内核版本号说明此内核是-stable 版本。它们包含内核的相对较小且至关重要的修补，这些修补针对安全性问题或者严重的内核退步。

这种版本的内核适用于那些期望获得最新的稳定版内核并且不想参与测试开发版或者实验版的用户。

稳定版内核树版本由“稳定版”小组（邮件地址 <[stable@vger.kernel.org](mailto:stable@vger.kernel.org)>）维护，一般隔周发布新版本。

内核源码中的 Documentation/process/stable-kernel-rules.rst 文件具体描述了可被稳定版内核接受的修改类型以及发布的流程。

## 报告 bug

[bugzilla.kernel.org](http://bugzilla.kernel.org) 是 Linux 内核开发者们用来跟踪内核 Bug 的网站。我们鼓励用户在这个工具中报告找到的所有 bug。如何使用内核 bugzilla 的细节请访问：

<http://test.kernel.org/bugzilla/faq.html>

内核源码主目录中的 :ref:`admin-guide/reporting-bugs.rst <reportingbugs>` 文件里有一个很好的模板。它指导用户如何报告可能的内核 bug 以及需要提供哪些信息来帮助内核开发者们找到问题的根源。

## 利用 bug 报告

练习内核开发技能的最好办法就是修改其他人报告的 bug。你不光可以帮助内核变得更加稳定，还可以学会如何解决实际问题从而提高自己的技能，并且让其他开发者感受到你的存在。修改 bug 是赢得其他开发者赞誉的最好办法，因为并不是很多人都喜欢浪费时间去修改别人报告的 bug。

要尝试修改已知的 bug，请访问 <http://bugzilla.kernel.org> 网址。

## 邮件列表

正如上面的文档所描述，大多数的骨干内核开发者都加入了 Linux Kernel 邮件列表。如何订阅和退订列表的细节可以在这里找到：

<http://vger.kernel.org/vger-lists.html#linux-kernel>

网上很多地方都有这个邮件列表的存档 (archive)。可以使用搜索引擎来找到这些存档。比如：

<http://dir.gmane.org/gmane.linux.kernel>

在发信之前，我们强烈建议你先在存档中搜索你想要讨论的问题。很多已经被详细讨论过的问题只在邮件列表的存档中可以找到。

大多数内核子系统也有自己独立的邮件列表来协调各自的开发工作。从 MAINTAINERS 文件中可以找到不同话题对应的邮件列表。

很多邮件列表架设在 kernel.org 服务器上。这些列表的信息可以在这里找到：

<http://vger.kernel.org/vger-lists.html>

在使用这些邮件列表时，请记住保持良好的行为习惯。下面的链接提供了与这些列表（或任何其它邮件列表）交流的一些简单规则，虽然内容有点滥竽充数。

<http://www.albion.com/netiquette/>

当有很多人回复你的邮件时，邮件的抄送列表会变得很长。请不要将任何人从抄送列表中删除，除非你有足够的理由这么做。也不要只回复到邮件列表。请习惯于同一封邮件接收两次（一封来自发送者一封来自邮件列表），而不要试图通过添加一些奇特的邮件头来解决这个问题，人们不会喜欢的。

记住保留你所回复内容的上下文和源头。在你回复邮件的顶部保留“某某某说到……”这几行。将你的评论加在被引用的段落之间而不要放在邮件的顶部。

如果你在邮件中附带补丁，请确认它们是可以直接阅读的纯文本（如*Documentation/translations/zh\_CN/process/submitting-patches.rst* 文档中所述）。内核开发者们不希望遇到附件或者被压缩了的补丁。只有这样才能保证他们可以直接评论你的每行代码。请确保你使用的邮件发送程序不会修改空格和制表符。一个防范性的测试方法是先将邮件发送给自己，然后自己尝试是否可以顺利地打上收到的补丁。如果测试不成功，请调整或者更换你的邮件发送程序直到它正确工作为止。

总而言之，请尊重其他的邮件列表订阅者。

## 同内核社区合作

内核社区的目标就是提供尽善尽美的内核。所以当你提交补丁期望被接受进内核的时候，它的技术价值以及其他方面都将被评审。那么你可能会得到什么呢？

- 批评
- 评论
- 要求修改
- 要求证明修改的必要性
- 沉默

要记住，这些是把补丁放进内核的正常情况。你必须学会听取对补丁的批评和评论，从技术层面评估它们，然后要么重写你的补丁要么简明扼要地论证修改是不必要的。如果你发的邮件没有得到任何回应，请过几天后再试一次，因为有时信件会淹没在茫茫信海中。

你不应该做的事情：

- 期望自己的补丁不受任何质疑就直接被接受
- 翻脸
- 忽略别人的评论
- 没有按照别人的要求做任何修改就重新提交

在一个努力追寻最好技术方案的社区里，对于一个补丁有多少好处总会有不同的见解。你必须要抱着合作的态度，愿意改变自己的观点来适应内核的风格。或者至少愿意去证明你的想法是有价值的。记住，犯错误是允许的，只要你愿意朝着正确的方案去努力。

如果你的第一个补丁换来的是一堆修改建议，这是很正常的。这并不代表你的补丁不会被接受，也不意味着有人和你作对。你只需要改正所有提出的问题然后重新发送你的补丁。

## 内核社区和公司文化的差异

内核社区的工作模式同大多数传统公司开发队伍的工作模式并不相同。下面这些例子，可以帮助你避免某些可能发生问题：用这些话介绍你的修改提案会有好处：（在任何时候你都不应该用中文写提案）

- 它同时解决了多个问题
- 它删除了 2000 行代码
- 这是补丁，它已经解释了我想要说明的
- 我在 5 种不同的体系结构上测试过它……
- 这是一系列小补丁用来……
- 这个修改提高了普通机器的性能……

应该避免如下的说法：

- 我们在 AIX/ptx/Solaris 就是这么做的，所以这么做肯定是好的……
- 我做这行已经 20 年了，所以……
- 为了我们公司赚钱考虑必须这么做
- 这是我们的企业产品线所需要的
- 这里是描述我观点的 1000 页设计文档
- 这是一个 5000 行的补丁用来……
- 我重写了现在乱七八糟的代码，这就是……
- 我被规定了最后期限，所以这个补丁需要立刻被接受

另外一个内核社区与大部分传统公司的软件开发队伍不同的地方是无法面对面地交流。使用电子邮件和 IRC 聊天工具做为主要沟通工具的一个好处是性别和种族歧视将会更少。Linux 内核的工作环境更能接受妇女和少数族群，因为每个人在别人眼里只是一个邮件地址。国际化也帮助了公平的实现，因为你无法通过姓名来判断人的性别。男人有可能叫李丽，女人也有可能叫王刚。大多数在 Linux 内核上工作过并表达过看法的女性对在 linux 上工作的经历都给出了正面的评价。

对于一些不习惯使用英语的人来说，语言可能是一个引起问题的障碍。在邮件列表中要正确地表达想法必需良好地掌握语言，所以建议你在发送邮件之前最好检查一下英文写得是否正确。

### 拆分修改

Linux 内核社区并不喜欢一下接收大段的代码。修改需要被恰当地介绍、讨论并且拆分成独立的小段。这几乎完全和公司中的习惯背道而驰。你的想法应该在开发最开始的阶段就让大家知道，这样你就可以及时获得对你正在进行的开发的反馈。这样也会让社区觉得你是在和他们协作，而不是仅仅把他们当作倾销新功能的对象。无论如何，你不要一次性地向邮件列表发送 50 封信，你的补丁序列应该永远用不到这么多。

将补丁拆开的原因如下：

- 1) 小的补丁更有可能被接受，因为它们不需要太多的时间和精力去验证其正确性。一个 5 行的补丁，可能在维护者看了一眼以后就会被接受。而 500 行的补丁则需要数个小时来审查其正确性（所需时间随补丁大小增加大约呈指数级增长）。

当出了问题的时候，小的补丁也会让调试变得非常容易。一个一个补丁地回溯将会比仔细剖析一个被打上的大补丁（这个补丁破坏了其他东西）容易得多。

### 2) 不光发送小的补丁很重要，在提交之前重新编排、化简（或者仅仅重新排列）补丁也是很重要的。

这里有内核开发者 **Al Viro** 打的一个比方：“想象一个老师正在给学生批改数学作业。老师并不希望看到学生为了得到正确解法所进行的尝试和产生的错误。他希望看到的是最干净最优雅的解答。好学生了解这点，绝不会把最终解决之前的中间方案提交上去。”

内核开发也是这样。维护者和评审者不希望看到一个人在解决问题时的思考过程。他们只希望看到简单和优雅的解决方案。

直接给出一流的解决方案，和社区一起协作讨论尚未完成的工作，这两者之间似乎很难找到一个平衡点。所以最好尽早开始收集有利于你进行改进的反馈；同时也要保证修改分成很多小块，这样在整个项目都准备好被包含进内核之前，其中的一部分可能会先被接收。

你必须明白这么做是无法令人接受的：试图将不完整的代码提交进内核，然后再找时间修复。

### 证明修改的必要性

除了将补丁拆成小块，很重要的一点是让 Linux 社区了解他们为什么需要这样修改。你必须证明新功能是有人需要的并且是有用的。

### 记录修改

当你发送补丁的时候，需要特别留意邮件正文的内容。因为这里的信息将会做为补丁的修改记录 (ChangeLog)，会被一直保留以备大家查阅。它需要完全地描述补丁，包括：

- 为什么需要这个修改
- 补丁的总体设计
- 实现细节
- 测试结果

想了解它具体应该看起来像什么, 请查阅以下文档中的 “**ChangeLog**” 章节:

“**The Perfect Patch**” <https://www.ozlabs.org/~akpm/stuff/tpp.txt>

这些事情有时候做起来很难。想要在任何方面都做到完美可能需要好几年时间。这是一个持续提高的过程, 它需要大量的耐心和决心。只要不放弃, 你一定可以做到。很多人已经做到了, 而他们都曾经和现在的你站在同样的起点上。

## 感谢

感谢 Paolo Ciarrocchi 允许 “开发流程” 部分基于他所写的文章 ([http://www.kerneltravel.net/newbie/2.6-development\\_process](http://www.kerneltravel.net/newbie/2.6-development_process)), 感谢 Randy Dunlap 和 Gerrit Huizenga 完善了应该说和不该说的列表。感谢 Pat Mochel, Hanna Linder, Randy Dunlap, Kay Sievers, Vojtech Pavlik, Jan Kara, Josh Boyer, Kees Cook, Andrew Morton, Andi Kleen, Vadim Lobanov, Jesper Juhl, Adrian Bunk, Keri Harris, Frans Pop, David A. Wheeler, Junio Hamano, Michael Kerrisk 和 Alex Shepard 的评审、建议和贡献。没有他们的帮助, 这篇文档是不可能完成的。

英文版维护者: Greg Kroah-Hartman <[greg@kroah.com](mailto:greg@kroah.com)>

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解, 而不是作为一个分支。因此, 如果您对此文件有任何意见或更新, 请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题, 请联系该文件的译者, 或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/process/code-of-conduct.rst

**Translator** Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

## 贡献者契约行为准则

### 我们的誓言

为了营造一个开放、友好的环境, 我们作为贡献者和维护人承诺, 让我们的社区和参与者, 拥有一个无骚扰的体验, 无论年龄、体型、残疾、种族、性别特征、性别认同和表达、经验水平、教育程度、社会状况, 经济地位、国籍、个人外貌、种族、宗教或性身份和取向。

### 我们的标准

有助于创造积极环境的行为包括:

- 使用欢迎和包容的语言
- 尊重不同的观点和经验
- 优雅地接受建设性的批评
- 关注什么对社区最有利
- 对其他社区成员表示同情

参与者的不可接受行为包括:

- 使用性意味的语言或意象以及不受欢迎的性注意或者更过分的行为
- 煽动、侮辱/贬损评论以及个人或政治攻击
- 公开或私下骚扰
- 未经明确许可，发布他人的私人信息，如物理或电子地址。
- 在专业场合被合理认为不适当的其他行为

### 我们的责任

维护人员负责澄清可接受行为的标准，并应针对任何不可接受行为采取适当和公平的纠正措施。

维护人员有权和责任删除、编辑或拒绝与本行为准则不一致的评论、承诺、代码、wiki 编辑、问题和其他贡献，或暂时或永久禁止任何贡献者从事他们认为不适当、威胁、冒犯或有害的其他行为。

### 范围

当个人代表项目或其社区时，本行为准则既适用于项目空间，也适用于公共空间。代表一个项目或社区的例子包括使用一个正式的项目电子邮件地址，通过一个正式的社交媒体帐户发布，或者在在线或离线事件中担任指定的代表。项目维护人员可以进一步定义和澄清项目的表示。

### 执行

如有滥用、骚扰或其他不可接受的行为，可联系行为准则委员会 <[conduct@kernel.org](mailto:conduct@kernel.org)>。所有投诉都将接受审查和调查，并将得到必要和适当的答复。行为准则委员会有义务对事件报告人保密。具体执行政策的进一步细节可单独公布。

## 归属

本行为准则改编自《贡献者契约》，版本 1.4，可从 <https://www.contributor-covenant.org/version/1/4/code-of-conduct.html> 获取。

## 解释

有关 Linux 内核社区如何解释此文档，请参阅 [Linux 内核贡献者契约行为准则解释](#)

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

**Original** Documentation/process/code-of-conduct-interpretation.rst

**Translator** Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

## Linux 内核贡献者契约行为准则解释

[贡献者契约行为准则](#) 准则是一个通用文档，旨在为几乎所有开源社区提供一套规则。每个开源社区都是独一无二的，Linux 内核也不例外。因此，本文描述了 Linux 内核社区中如何解释它。我们也不希望这种解释随着时间的推移是静态的，并将根据需要进行调整。

与开发软件的“传统”方法相比，Linux 内核开发工作是一个非常个人化的过程。你的贡献和背后的想法将被仔细审查，往往导致批判和批评。审查将几乎总是需要改进，材料才能包括在内核中。要知道这是因为所有相关人员都希望看到 Linux 整体成功的最佳解决方案。这个开发过程已经被证明可以创建有史以来最健壮的操作系统内核，我们不想做任何事情来导致提交质量和最终结果的下降。

## 维护者

行为准则多次使用“维护者”一词。在内核社区中，“维护者”是负责子系统、驱动程序或文件的任何人，并在内核源代码树的维护者文件中列出。

### 责任

《行为准则》提到了维护人员的权利和责任，这需要进一步澄清。

首先，最重要的是，有一个合理的期望是由维护人员通过实例来领导。

也就是说，我们的社区是广阔的，对维护者没有新的要求，他们单方面处理其他人在他们活跃的社区的行为。这一责任由我们所有人承担，最终《行为准则》记录了最终的上诉路径，以防有关行为问题的问题悬而未决。

维护人员应该愿意在出现问题时提供帮助，并在需要时与社区中的其他人合作。如果您不确定如何处理出现的情况，请不要害怕联系技术咨询委员会 (TAB) 或其他维护人员。除非您愿意，否则不会将其视为违规报告。如果您不确定是否该联系 TAB 或任何其他维护人员，请联系我们的冲突调解人 Mishi Choudhary <[mishi@linux.com](mailto:mishi@linux.com)>。

最后，“善待对方”才是每个人的最终目标。我们知道每个人都是人，有时我们都会失败，但我们所有人的首要目标应该是努力友好地解决问题。执行行为准则将是最后的选择。

我们的目标是创建一个强大的、技术先进的操作系统，以及所涉及的技术复杂性，这自然需要专业知识和决策。所需的专业知识因贡献领域而异。它主要由上下文和技术复杂性决定，其次由贡献者和维护者的期望决定。

专家的期望和决策都要经过讨论，但在最后，为了取得进展，必须能够做出决策。这一特权掌握在维护人员和项目领导的手中，预计将善意使用。

因此，设定专业知识期望、作出决定和拒绝不适当的贡献不被视为违反行为准则。

虽然维护人员一般都欢迎新来者，但他们帮助（新）贡献者克服障碍的能力有限，因此他们必须确定优先事项。这也不应被视为违反了行为准则。内核社区意识到这一点，并以各种形式提供入门级节目，如 [kernelnewbies.org](http://kernelnewbies.org)。

### 范围

Linux 内核社区主要在一组公共电子邮件列表上进行交互，这些列表分布在由多个不同公司或个人控制的多个不同服务器上。所有这些列表都在内核源代码树中的 MAINTAINERS 文件中定义。发送到这些邮件列表的任何电子邮件都被视为包含在行为准则中。

使用 kernel.org bugzilla 和其他子系统 bugzilla 或 bug 跟踪工具的开发人员应该遵循行为准则的指导原则。Linux 内核社区没有“官方”项目电子邮件地址或“官方”社交媒体地址。使用 kernel.org 电子邮件帐户执行的任何活动必须遵循为 kernel.org 发布的行为准则，就像任何使用公司电子邮件帐户的个人必须遵循该公司的特定规则一样。

行为准则并不禁止在邮件列表消息、内核更改日志消息或代码注释中继续包含名称、电子邮件地址和相关注释。其他论坛中的互动包括在适用于上述论坛的任何规则中，通常不包括在行为准则中。除了在极端情况下可考虑的例外情况。

提交给内核的贡献应该使用适当的语言。在行为准则之前已经存在的内容现在不会被视为违反。然而，不适当的语言可以被视为一个 bug；如果任何相关方提交补丁，这样的 bug 将被更快地修复。当前属于用户/内核 API 的一部分的表达式，或者反映已发布标准或规范中使用的术语的表达式，不被视为 bug。

## 执行

行为准则中列出的地址属于行为准则委员会。<https://kernel.org/code-of-conduct.html> 列出了在任何给定时间接收这些电子邮件的确切成员。成员不能访问在加入委员会之前或离开委员会之后所做的报告。

最初的行为准则委员会由 TAB 的志愿者以及作为中立第三方的专业调解人组成。委员会的首要任务是建立文件化的流程，并将其公开。

如果报告人不希望将整个委员会纳入投诉或关切，可直接联系委员会的任何成员，包括调解人。

行为准则委员会根据流程审查案例（见上文），并根据需要和适当与 TAB 协商，例如请求和接收有关内核社区的信息。

委员会做出的任何决定都将提交到表中，以便在必要时与相关维护人员一起执行。行为准则委员会的决定可以通过三分之二的投票推翻。

每季度，行为准则委员会和标签将提供一份报告，概述行为准则委员会收到的匿名报告及其状态，以及任何否决决定的细节，包括完整和可识别的投票细节。

我们希望在启动期之后为行为准则委员会人员配备建立一个不同的流程。发生此情况时，将使用该信息更新此文档。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

## Original Documentation/process/submitting-patches.rst

译者：

中文版维护者： 钟宇 TripleX Chung <xxx.phy@gmail.com>  
中文版翻译者： 钟宇 TripleX Chung <xxx.phy@gmail.com>  
时奎亮 Alex Shi <alex.shi@linux.alibaba.com>  
中文版校译者： 李阳 Li Yang <leoyang.li@nxp.com>  
王聪 Wang Cong <xifyou.wangcong@gmail.com>

### 如何让你的改动进入内核

对于想要将改动提交到 Linux 内核的个人或者公司来说，如果不熟悉“规矩”，提交的流程会让人畏惧。本文档收集了一系列建议，这些建议可以大大的提高你的改动被接受的机会。

以下文档含有大量简洁的建议，具体请见：Documentation/process 同样，[Documentation/translations/zh\\_CN/process/submit-checklist.rst](#) 给出在提交代码前需要检查的项目的列表。如果你在提交一个驱动程序，那么同时阅读一下：[Documentation/process/submitting-drivers.rst](#)

其中许多步骤描述了 Git 版本控制系统的默认行为；如果您使用 Git 来准备补丁，您将发现它为您完成的大部分机械工作，尽管您仍然需要准备和记录一组合理的补丁。一般来说，使用 git 将使您作为内核开发人员的生活更轻松。

### 0) 获取当前源码树

如果您没有一个可以使用当前内核源代码的存储库，请使用 git 获取一个。您将要从主线存储库开始，它可以通过以下方式获取：

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

但是，请注意，您可能不希望直接针对主线树进行开发。大多数子系统维护人员运行自己的树，并希望看到针对这些树准备的补丁。请参见 MAINTAINERS 文件中子系统的 **T:** 项以查找该树，或者简单地询问维护者该树是否未在其中列出。

仍然可以通过 tarballs 下载内核版本（如下一节所述），但这是进行内核开发的一种困难的方式。

### 1) “diff -up”

使用 “diff -up” 或者 “diff -uprN” 来创建补丁。

所有内核的改动，都是以补丁的形式呈现的，补丁由 diff(1) 生成。创建补丁的时候，要确认它是以“unified diff”格式创建的，这种格式由 diff(1) 的 ‘-u’ 参数生成。而且，请使用 ‘-p’ 参数，那样会显示每个改动所在的 C 函数，使得产生的补丁容易读得多。补丁应该基于内核源代码树的根目录，而不是里边的任何子目录。

为一个单独的文件创建补丁，一般来说这样做就够了：

```
SRCTREE=linux
MYFILE=drivers/net/mydriver.c

cd $SRCTREE
cp $MYFILE $MYFILE.orig
vi $MYFILE      # make your change
cd ..
diff -up $SRCTREE/$MYFILE{.orig,} > /tmp/patch
```

为多个文件创建补丁，你可以解开一个没有修改过的内核源代码树，然后和你自己的代码树之间做 diff。例如：

```
MYSRC=/devel/Linux

tar xvfz linux-3.19.tar.gz
mv linux-3.19 linux-3.19-vanilla
diff -uprN -X linux-3.19-vanilla/Documentation/dontdiff \
    linux-3.19-vanilla $MYSRC > /tmp/patch
```

“`dontdiff`”是内核在编译的时候产生的文件的列表，列表中的文件在 `diff(1)` 产生的补丁里会被跳过。

确定你的补丁里没有包含任何不属于这次补丁提交的额外文件。记得在用 `diff(1)` 生成补丁之后，审阅一次补丁，以确保准确。

如果你的改动很散乱，你应该研究一下如何将补丁分割成独立的部分，将改动分割成一系列合乎逻辑的步骤。这样更容易让其他内核开发者审核，如果你想你的补丁被接受，这是很重要的。请参阅：[3\) 拆分你的改动](#)

如果你用 `git`, `git rebase -i` 可以帮助你这一点。如果你不用 `git, quilt <https://savannah.nongnu.org/projects/quilt>` 另外一个流行的选择。

## 2) 描述你的改动

描述你的问题。无论您的补丁是一行错误修复还是 5000 行新功能，都必须有一个潜在的问题激励您完成这项工作。让审稿人相信有一个问题值得解决，让他们读完第一段是有意义的。

描述用户可见的影响。直接崩溃和锁定是相当有说服力的，但并不是所有的错误都那么明目张胆。即使在代码审查期间发现了这个问题，也要描述一下您认为它可能对用户产生的影响。请记住，大多数 Linux 安装运行的内核来自二级稳定树或特定于供应商/产品的树，只从上游精选特定的补丁，因此请包含任何可以帮助您将更改定位到下游的内容：触发的场景、DMESG 的摘录、崩溃描述、性能回归、延迟尖峰、锁定等。

量化优化和权衡。如果您声称在性能、内存消耗、堆栈占用空间或二进制大小方面有所改进，请包括支持它们的数字。但也要描述不明显的成本。优化通常不是免费的，而是在 CPU、内存和可读性之间进行权衡；或者，探索性的工作，在不同的工作负载之间进行权衡。请描述优化的预期缺点，以便审阅者可以权衡成本和收益。

一旦问题建立起来，就要详细地描述一下您实际在做什么。对于审阅者来说，用简单的英语描述代码的变化是很重要的，以验证代码的行为是否符合您的意愿。

如果您将补丁描述写在一个表单中，这个表单可以很容易地作为“提交日志”放入 Linux 的源代码管理系统 `git` 中，那么维护人员将非常感谢您。见[15\) 明确回复邮件头 \(In-Reply-To\)](#).

每个补丁只解决一个问题。如果你的描述开始变长，这就表明你可能需要拆分你的补丁。请见[3\) 拆分你的改动](#) 提交或重新提交修补程序或修补程序系列时，请包括完整的修补程序说明和理由。不要只说这是补丁（系列）的第几版。不要期望子系统维护人员引用更早的补丁版本或引用 URL 来查找补丁描述并将其放入补丁中。也就是说，补丁（系列）及其描述应该是独立的。这对维护人员和审查人员都有好处。一些评审者可能甚至没有收到补丁的早期版本。

描述你在命令语气中的变化，例如“`make xyzzy do frotz`”而不是“[This patch]`make xyzzy do frotz`”或“[I]changed xyzzy to do frotz”，就好像你在命令代码库改变它的行为一样。

如果修补程序修复了一个记录的 bug 条目, 请按编号和 URL 引用该 bug 条目。如果补丁来自邮件列表讨论, 请给出邮件列表存档的 URL; 使用带有 Message-ID 的 <https://lore.kernel.org/> 重定向, 以确保链接不会过时。

但是, 在没有外部资源的情况下, 尽量让你的解释可理解。除了提供邮件列表存档或 bug 的 URL 之外, 还要总结需要提交补丁的相关讨论要点。

如果您想要引用一个特定的提交, 不要只引用提交的 SHA-1 ID。还请包括提交的一行摘要, 以便于审阅者了解它是关于什么的。例如:

```
Commit e21d2170f36602ae2708 ("video: remove unnecessary  
platform_set_drvdata()") removed the unnecessary  
platform_set_drvdata(), but left the variable "dev" unused,  
delete it.
```

您还应该确保至少使用前 12 位 SHA-1 ID. 内核存储库包含 \*许多\* 对象, 使与较短的 ID 发生冲突的可能性很大。记住, 即使现在不会与您的六个字符 ID 发生冲突, 这种情况可能五年后改变。

如果修补程序修复了特定提交中的错误, 例如, 使用 `git bisect`, 请使用带有前 12 个字符 SHA-1 ID 的“Fixes:”标记和单行摘要。为了简化不要将标记拆分为多个, 行、标记不受分析脚本“75 列换行”规则的限制。例如:

```
Fixes: 54a4f0239f2e ("KVM: MMU: make kvm_mmu_zap_page() return the number of pages it actually  
→ freed")
```

下列 `git config` 设置可以添加让 `git log`, `git show` 漂亮的显示格式:

```
[core]  
    abbrev = 12  
[pretty]  
    fixes = Fixes: %h (%s)
```

### 3) 拆分你的改动

将每个逻辑更改分隔成一个单独的补丁。

例如, 如果你的改动里同时有 bug 修正和性能优化, 那么把这些改动拆分到两个或者更多的补丁文件中。如果你的改动包含对 API 的修改, 并且修改了驱动程序来适应这些新的 API, 那么把这些修改分成两个补丁。

另一方面, 如果你将一个单独的改动做成多个补丁文件, 那么将它们合并成一个单独的补丁文件。这样一个逻辑上单独的改动只被包含在一个补丁文件里。

如果有一个补丁依赖另外一个补丁来完成它的改动, 那没问题。简单的在你的补丁描述里指出“这个补丁依赖某补丁”就好了。

在将您的更改划分为一系列补丁时, 要特别注意确保内核在系列中的每个补丁之后都能正常构建和运行。使用 `git bisect` 来追踪问题的开发者可能会在任何时候分割你的补丁系列; 如果你在中间引入错误, 他们不会感谢你。

如果你不能将补丁浓缩成更少的文件，那么每次大约发送出 15 个，然后等待审查和集成。

#### 4) 检查你的更改风格

检查您的补丁是否存在基本样式冲突，详细信息可在 [Documentation/translations/zh\\_CN/process/coding-style.rst](#) 中找到。如果不这样做，只会浪费审稿人的时间，并且会导致你的补丁被拒绝，甚至可能没有被阅读。

一个重要的例外是在将代码从一个文件移动到另一个文件时——在这种情况下，您不应该在移动代码的同一个补丁中修改移动的代码。这清楚地描述了移动代码和您的更改的行为。这大大有助于审查实际差异，并允许工具更好地跟踪代码本身的历史。

在提交之前，使用补丁样式检查程序检查补丁（scripts/check patch.pl）。不过，请注意，样式检查程序应该被视为一个指南，而不是作为人类判断的替代品。如果您的代码看起来更好，但有违规行为，那么最好不要使用它。

检查者报告三个级别：

- ERROR：很可能出错的事情
- WARNING：需要仔细审查的事项
- CHECK：需要思考的事情

您应该能够判断您的补丁中存在的所有违规行为。

#### 5) 选择补丁收件人

您应该总是在任何补丁上复制相应的子系统维护人员，以获得他们维护的代码；查看维护人员文件和源代码修订历史记录，以了解这些维护人员是谁。脚本 scripts/get\_Maintainer.pl 在这个步骤中非常有用。如果您找不到正在工作的子系统的维护人员，那么 Andrew Morton ([akpm@linux-foundation.org](mailto:akpm@linux-foundation.org)) 将充当最后的维护人员。

您通常还应该选择至少一个邮件列表来接收补丁集的。[linux-kernel@vger.kernel.org](mailto:linux-kernel@vger.kernel.org) 作为最后一个解决办法的列表，但是这个列表上的体积已经引起了许多开发人员的拒绝。在 MAINTAINERS 文件中查找子系统特定的列表；您的补丁可能会在那里得到更多的关注。不过，请不要发送垃圾邮件到无关的列表。

许多与内核相关的列表托管在 vger.kernel.org 上；您可以在 <http://vger.kernel.org/vger-lists.html> 上找到它们的列表。不过，也有与内核相关的列表托管在其他地方。

不要一次发送超过 15 个补丁到 vger 邮件列表!!!!

Linus Torvalds 是决定改动能否进入 Linux 内核的最终裁决者。他的 e-mail 地址是 <[torvalds@linux-foundation.org](mailto:torvalds@linux-foundation.org)>。他收到的 e-mail 很多，所以一般的说，最好别给他发 e-mail。

如果您有修复可利用安全漏洞的补丁，请将该补丁发送到 [security@kernel.org](mailto:security@kernel.org)。对于严重的 bug，可以考虑短期暂停以允许分销商向用户发布补丁；在这种情况下，显然不应将补丁发送到任何公共列表。

修复已发布内核中严重错误的补丁程序应该指向稳定版维护人员，方法是放这样的一行：

Cc: stable@vger.kernel.org

进入补丁的签准区（注意，不是电子邮件收件人）。除了这个文件之外，您还应该阅读 Documentation/process/stable-kernel-rules.rst

但是，请注意，一些子系统维护人员希望得出他们自己的结论，即哪些补丁应该被放到稳定的树上。尤其是网络维护人员，不希望看到单个开发人员在补丁中添加像上面这样的行。

如果更改影响到用户和内核接口，请向手册页维护人员（如维护人员文件中所列）发送手册页补丁，或至少发送更改通知，以便一些信息进入手册页。还应将用户空间 API 更改复制到 linux-api@vger.kernel.org。

### 6) 没有 MIME 编码，没有链接，没有压缩，没有附件，只有纯文本

Linus 和其他的内核开发者需要阅读和评论你提交的改动。对于内核开发者来说，可以“引用”你的改动很重要，使用一般的 e-mail 工具，他们就可以在你的代码的任何位置添加评论。

因为这个原因，所有的提交的补丁都是 e-mail 中“内嵌”的。

**Warning:** 如果你使用剪切-粘贴你的补丁，小心你的编辑器的自动换行功能破坏你的补丁

不要将补丁作为 MIME 编码的附件，不管是否压缩。很多流行的 e-mail 软件不是任何时候都将 MIME 编码的附件当作纯文本发送的，这会使得别人无法在你的代码中加评论。另外，MIME 编码的附件会让 Linus 多花一点时间来处理，这就降低了你的改动被接受的可能性。

例外：如果你的邮递员弄坏了补丁，那么有人可能会要求你使用 mime 重新发送补丁

请参阅 Documentation/translations/zh\_CN/process/email-clients.rst 以获取有关配置电子邮件客户端以使其不受影响地发送修补程序的提示。

### 7) e-mail 的大小

大的改动对邮件列表不合适，对某些维护者也不合适。如果你的补丁，在不压缩的情况下，超过了 300kB，那么你最好将补丁放在一个能通过 internet 访问的服务器上，然后用指向你的补丁的 URL 替代。但是请注意，如果您的补丁超过了 300kb，那么它几乎肯定需要被破坏。

### 8) 回复评审意见

你的补丁几乎肯定会得到评审者对补丁改进方法的评论。您必须对这些评论作出回应；让补丁被忽略的一个好办法就是忽略审阅者的意见。不会导致代码更改的意见或问题几乎肯定会带来注释或变更日志的改变，以便下一个评审者更好地了解正在发生的事情。

一定要告诉审稿人你在做什么改变，并感谢他们的时间。代码审查是一个累人且耗时的过程，审查人员有时会变得暴躁。即使在这种情况下，也要礼貌地回应并解决他们指出的问题。

## 9) 不要泄气或不耐烦

提交更改后, 请耐心等待。审阅者是忙碌的人, 可能无法立即访问您的修补程序。

曾几何时, 补丁曾在没有评论的情况下消失在空白中, 但开发过程比现在更加顺利。您应该在一周左右的时间内收到评论; 如果没有收到评论, 请确保您已将补丁发送到正确的位置。在重新提交或联系审阅者之前至少等待一周-在诸如合并窗口之类的繁忙时间可能更长。

## 10) 主题中包含 PATCH

由于到 linus 和 linux 内核的电子邮件流量很高, 通常会在主题行前面加上 [PATCH] 前缀. 这使 Linus 和其他内核开发人员更容易将补丁与其他电子邮件讨论区分开。

## 11) 签署你的作品-开发者原始认证

为了加强对谁做了何事的追踪, 尤其是对那些透过好几层的维护者的补丁, 我们建议在发送出去的补丁上加一个“sign-off” 的过程。

“sign-off” 是在补丁的注释的最后的简单的一行文字, 认证你编写了它或者其他人有权力将它作为开放源代码的补丁传递。规则很简单: 如果你能认证如下信息:

### 开发者来源证书 1.1

对于本项目的贡献, 我认证如下信息:

- (a) 这些贡献是完全或者部分的由我创建, 我有权利以文件中指出 的开放源代码许可证提交它; 或者
- (b) 这些贡献基于以前的工作, 据我所知, 这些以前的工作受恰当的开放 源代码许可证保护, 而且, 根据许可证, 我有权提交修改后的贡献, 无论是完全还是部分由我创造, 这些贡献都使用同一个开放源代码许可证 (除非我被允许用其它的许可证), 正如文件中指出的; 或者
- (c) 这些贡献由认证 (a), (b) 或者 (c) 的人直接提供给我, 而 且我没有修改它。
- (d) 我理解并同意这个项目和贡献是公开的, 贡献的记录 (包括我 一起提交的个人记录, 包括 sign-off ) 被永久维护并且可以和这个项目或者开放源代码的许可证同步地再发行。

那么加入这样一行:

Signed-off-by: Random J Developer <random@developer.example.org>

使用你的真名 (抱歉, 不能使用假名或者匿名。)

有人在最后加上标签。现在这些东西会被忽略, 但是你可以这样做, 来标记公司内部的过程, 或者只是指出关于 sign-off 的一些特殊细节。

如果您是子系统或分支维护人员，有时需要稍微修改收到的补丁，以便合并它们，因为树和提交者中的代码不完全相同。如果你严格遵守规则 (c)，你应该要求提交者重新发布，但这完全是在浪费时间和精力。规则 (b) 允许您调整代码，但是更改一个提交者的代码并让他认可您的错误是非常不礼貌的。要解决此问题，建议在最后一个由签名行和您的行之间添加一行，指示更改的性质。虽然这并不是强制性的，但似乎在描述前加上您的邮件和/或姓名（全部用方括号括起来），这足以让人注意到您对最后一分钟的更改负有责任。例如：

```
Signed-off-by: Random J Developer <random@developer.example.org>
[lucky@maintainer.example.org: struct foo moved from foo.c to foo.h]
Signed-off-by: Lucky K Maintainer <lucky@maintainer.example.org>
```

如果您维护一个稳定的分支机构，同时希望对作者进行致谢、跟踪更改、合并修复并保护提交者不受投诉，那么这种做法尤其有用。请注意，在任何情况下都不能更改作者的 ID (From 头)，因为它是出现在更改日志中的标识。

对回合 (back-porters) 的特别说明：在提交消息的顶部（主题行之后）插入一个补丁的起源指示似乎是一种常见且有用的实践，以便于跟踪。例如，下面是我们在 3.x 稳定版本中看到的内容：

```
Date: Tue Oct 7 07:26:38 2014 -0400

libata: Un-break ATA blacklist

commit 1c40279960bcd7d52dbdf1d466b20d24b99176c8 upstream.
```

还有，这里是一个旧版内核中的一个回合补丁：

```
Date: Tue May 13 22:12:27 2008 +0200

wireless, airo: waitbusy() won't delay

[backport of 2.6 commit b7acbd1f277c1eb23f344f899cfa4cd0bf36a]
```

## 12) 何时使用 Acked-by:, CC:, 和 Co-Developed by:

Singed-off-by: 标记表示签名者参与了补丁的开发，或者他/她在补丁的传递路径中。

如果一个人没有直接参与补丁的准备或处理，但希望表示并记录他们对补丁的批准，那么他们可以要求在补丁的变更日志中添加一个 Acked-by:

Acked-by: 通常由受影响代码的维护者使用，当该维护者既没有贡献也没有转发补丁时。

Acked-by: 不像签字人那样正式。这是一个记录，确认人至少审查了补丁，并表示接受。因此，补丁合并有时会手动将 Acker 的 “Yep, looks good to me” 转换为 Acked-By: (但请注意，通常最好要求一个明确的 Ack)。

Acked-by: 不一定表示对整个补丁的确认。例如，如果一个补丁影响多个子系统，并且有一个：来自一个子系统维护者，那么这通常表示只确认影响维护者代码的部分。这里应该仔细判断。如有疑问，应参考邮件列表档案中的原始讨论。

如果某人有机会对补丁进行评论，但没有提供此类评论，您可以选择在补丁中添加 Cc：这是唯一一个标签，它可以在没有被它命名的人显式操作的情况下添加，但它应该表明这个人是在补丁上抄送的。讨论中包含了潜在利益相关方。

**Co-developed-by:** 声明补丁是由多个开发人员共同创建的；当几个人在一个补丁上工作时，它用于将属性赋予共同作者（除了 From：所赋予的作者之外）。因为 Co-developed-by：表示作者身份，所以每个共同开发者：必须紧跟在相关合作作者的签名之后。标准的签核程序要求：标记的签核顺序应尽可能反映补丁的时间历史，而不管作者是通过 From：还是由 Co-developed-by：共同开发的。值得注意的是，最后一个签字人：必须始终是提交补丁的开发人员。

注意，当作者也是电子邮件标题“发件人：”行中列出的人时，“From：”标记是可选的。

作者提交的补丁程序示例：

```
<changelog>

Co-developed-by: First Co-Author <first@coauthor.example.org>
Signed-off-by: First Co-Author <first@coauthor.example.org>
Co-developed-by: Second Co-Author <second@coauthor.example.org>
Signed-off-by: Second Co-Author <second@coauthor.example.org>
Signed-off-by: From Author <from@author.example.org>
```

合作开发者提交的补丁示例：

```
From: From Author <from@author.example.org>

<changelog>

Co-developed-by: Random Co-Author <random@coauthor.example.org>
Signed-off-by: Random Co-Author <random@coauthor.example.org>
Signed-off-by: From Author <from@author.example.org>
Co-developed-by: Submitting Co-Author <sub@coauthor.example.org>
Signed-off-by: Submitting Co-Author <sub@coauthor.example.org>
```

### 13) 使用报告人：、测试人：、审核人：、建议人：、修复人：

**Reported-by:** 给那些发现错误并报告错误的人致谢，它希望激励他们在将来再次帮助我们。请注意，如果 bug 是以私有方式报告的，那么在使用 Reported-by 标记之前，请先请求权限。

**Tested-by:** 标记表示补丁已由指定的人（在某些环境中）成功测试。这个标签通知维护人员已经执行了一些测试，为将来的补丁提供了一种定位测试人员的方法，并确保测试人员的信誉。

**Reviewed-by:** 相反，根据审查人的声明，表明该补丁已被审查并被认为是可接受的：

### 审查人的监督声明

通过提供我的 Reviewed-by，我声明：

- (a) 我已经对这个补丁进行了一次技术审查，以评估它是否适合被包含到主线内核中。
- (b) 与补丁相关的任何问题、顾虑或问题都已反馈给提交者。我对提交者对我的评论的回应感到满意。
- (c) 虽然这一提交可能会改进一些东西，但我相信，此时，(1) 对内核进行了有价值的修改，(2) 没有包含争论中涉及的已知问题。
- (d) 虽然我已经审查了补丁并认为它是健全的，但我不会（除非另有明确说明）作出任何保证或保证它将在任何给定情况下实现其规定的功能或正常运行。

Reviewed-by 是一种观点声明，即补丁是对内核的适当修改，没有任何遗留的严重技术问题。任何感兴趣的审阅者（完成工作的人）都可以为一个补丁提供一个 Review-by 标签。此标签用于向审阅者提供致谢，并通知维护者已在修补程序上完成的审阅程度。Reviewed-by：当由已知了解主题区域并执行彻底检查的审阅者提供时，通常会增加补丁进入内核的可能性。

Suggested-by：表示补丁的想法是由指定的人提出的，并确保将此想法归功于指定的人。请注意，未经许可，不得添加此标签，特别是如果该想法未在公共论坛上发布。这就是说，如果我们勤快地致谢我们的创意者，他们很有希望在未来得到鼓舞，再次帮助我们。

Fixes：指示补丁在以前的提交中修复了一个问题。它可以很容易地确定错误的来源，这有助于检查错误修复。这个标记还帮助稳定内核团队确定应该接收修复的稳定内核版本。这是指示补丁修复的错误的首选方法。请参阅[2\) 描述你的改动](#) 描述您的更改以了解更多详细信息。

## 12) 标准补丁格式

本节描述如何格式化补丁本身。请注意，如果您的补丁存储在 Git 存储库中，则可以使用 git format-patch 进行正确的补丁格式设置。但是，这些工具无法创建必要的文本，因此请务必阅读下面的说明。

标准的补丁，标题行是：

```
Subject: [PATCH 001/123] 子系统：一句话概述
```

标准补丁的信体存在如下部分：

- 一个“from”行指出补丁作者。后跟空行（仅当发送修补程序的人不是作者时才需要）。
- 解释的正文，行以 75 列包装，这将被复制到永久变更日志来描述这个补丁。
- 一个空行
- 上面描述的“Signed-off-by”行，也将出现在更改日志中。
- 只包含 --- 的标记线。
- 任何其他不适合放在变更日志的注释。
- 实际补丁（diff 输出）。

标题行的格式，使得对标题行按字母序排序非常的容易 - 很多 e-mail 客户端都可以支持 - 因为序列号是用零填充的，所以按数字排序和按字母排序是一样的。

e-mail 标题中的“子系统”标识哪个内核子系统将被打补丁。

e-mail 标题中的“一句话概述”扼要的描述 e-mail 中的补丁。“一句话概述”不应该是一个文件名。对于一个补丁系列（“补丁系列”指一系列的多个相关补丁），不要对每个补丁都使用同样的“一句话概述”。

记住 e-mail 的“一句话概述”会成为该补丁的全局唯一标识。它会蔓延到 git 的改动记录里。然后“一句话概述”会被用在开发者的讨论里，用来指代这个补丁。用户将希望通过 google 来搜索“一句话概述”来找到那些讨论这个补丁的文章。当人们在两三个月后使用诸如 `gitk` 或 `git log --oneline` 之类的工具查看数千个补丁时，也会很快看到它。

出于这些原因，概述必须不超过 70-75 个字符，并且必须描述补丁的更改以及为什么需要补丁。既要简洁又要描述性很有挑战性，但写得好的概述应该这样做。

概述的前缀可以用方括号括起来：“Subject: [PATCH <tag>…] <概述>”。标记不被视为概述的一部分，而是描述应该如何处理补丁。如果补丁的多个版本已发送出来以响应评审（即“v1, v2, v3”）或“rfc”，以指示评审请求，那么通用标记可能包括版本描述符。如果一个补丁系列中有四个补丁，那么各个补丁可以这样编号：1/4、2/4、3/4、4/4。这可以确保开发人员了解补丁应用的顺序，并且他们已经查看或应用了补丁系列中的所有补丁。

一些标题的例子：

```
Subject: [patch 2/5] ext2: improve scalability of bitmap searching
Subject: [PATCHv2 001/207] x86: fix eflags tracking
```

“From”行是信体里的最上面一行，具有如下格式： From: Patch Author <[author@example.com](mailto:author@example.com)>

“From”行指明在永久改动日志里，谁会被确认为作者。如果没有“From”行，那么邮件头里的“From:”行会被用来决定改动日志中的作者。

说明的主题将会被提交到永久的源代码改动日志里，因此对那些早已经不记得和这个补丁相关的讨论细节的有能力的读者来说，是有意义的。包括补丁程序定位错误的（内核日志消息、OOPS 消息等）症状，对于搜索提交日志以寻找适用补丁的人尤其有用。如果一个补丁修复了一个编译失败，那么可能不需要包含所有编译失败；只要足够让搜索补丁的人能够找到它就行了。与概述一样，既要简洁又要描述性。

“—”标记行对于补丁处理工具要找到哪里是改动日志信息的结束，是不可缺少的。

对于“—”标记之后的额外注解，一个好的用途就是用来写 `diffstat`，用来显示修改了什么文件和每个文件都增加和删除了多少行。`diffstat` 对于比较大的补丁特别有用。其余那些只是和时刻或者开发者相关的注解，不合适放到永久的改动日志里的，也应该放这里。使用 `diffstat` 的选项 “`-p 1 -w 70`” 这样文件名就会从内核源代码树的目录开始，不会占用太宽的空间（很容易适合 80 列的宽度，也许会有一些缩进。）

在后面的参考资料中能看到适当的补丁格式的更多细节。

## 15) 明确回复邮件头 (In-Reply-To)

手动添加回复补丁的标题头 (In-Reply-To:) 是有帮助的（例如，使用 `git send-email`）将补丁与以前的相关讨论关联起来，例如，将 bug 修复程序链接到电子邮件和 bug 报告。但是，对于多补丁系列，最好避免在回复时使用链接到该系列的旧版本。这样，补丁的多个版本就不会成为电子邮件客户端中无法管理的引用序列。如果链接有用，可以使用 <https://lore.kernel.org/> 重定向器（例如，在封面电子邮件文本中）链接到补丁系列的早期版本。

## 16) 发送 git pull 请求

如果您有一系列补丁，那么让维护人员通过 `git pull` 操作将它们直接拉入子系统存储库可能是最方便的。但是，请注意，从开发人员那里获取补丁比从邮件列表中获取补丁需要更高的信任度。因此，许多子系统维护人员不愿意接受请求，特别是来自新的未知开发人员的请求。如果有疑问，您可以在封面邮件中使用 `pull` 请求作为补丁系列正常发布的一个选项，让维护人员可以选择使用其中之一。

`pull` 请求的主题行中应该有 [Git Pull]。请求本身应该在一行中包含存储库名称和感兴趣的分支；它应该看起来像：

```
Please pull from  
git://jdelvare.pck.nerim.net/jdelvare-2.6 i2c-for-linus  
to get these changes:
```

`pull` 请求还应该包含一条整体消息，说明请求中将包含什么，一个补丁本身的 `Git shortlog` 以及一个显示补丁系列整体效果的 `diffstat`。当然，将所有这些信息收集在一起的最简单方法是让 `git` 使用 `git request-pull` 命令为您完成这些工作。

一些维护人员（包括 Linus）希望看到来自已签名提交的请求；这增加了他们对你的请求信心。特别是，在没有签名标签的情况下，Linus 不会从像 Github 这样的公共托管站点拉请求。

创建此类签名的第一步是生成一个 GNRPG 密钥，并由一个或多个核心内核开发人员对其进行签名。这一步对新开发人员来说可能很困难，但没有办法绕过它。参加会议是找到可以签署您的密钥的开发人员的好方法。

一旦您在 Git 中准备了一个您希望有人拉的补丁系列，就用 `git tag -s` 创建一个签名标记。这将创建一个新标记，标识该系列中的最后一次提交，并包含用您的私钥创建的签名。您还可以将 `changelog` 样式的消息添加到标记中；这是一个描述拉请求整体效果的理想位置。

如果维护人员将要从中提取的树不是您正在使用的存储库，请不要忘记将已签名的标记显式推送到公共树。

生成拉请求时，请使用已签名的标记作为目标。这样的命令可以实现：

```
git request-pull master git://my.public.tree/linux.git my-signed-tag
```

## 参考文献

**Andrew Morton, “The perfect patch” (tpp).** <<https://www.ozlabs.org/~akpm/stuff/tpp.txt>>

**Jeff Garzik, “Linux kernel patch submission format” .** <<https://web.archive.org/web/20180829112450/http://linux.yyz.us/patch-format.html>>

**Greg Kroah-Hartman, “How to piss off a kernel subsystem maintainer” .** <<http://www.kroah.com/log/linux/maintainer.html>>  
[<http://www.kroah.com/log/linux/maintainer-02.html>](http://www.kroah.com/log/linux/maintainer-02.html)  
[<http://www.kroah.com/log/linux/maintainer-03.html>](http://www.kroah.com/log/linux/maintainer-03.html)  
[<http://www.kroah.com/log/linux/maintainer-04.html>](http://www.kroah.com/log/linux/maintainer-04.html)  
[<http://www.kroah.com/log/linux/maintainer-05.html>](http://www.kroah.com/log/linux/maintainer-05.html)  
[<http://www.kroah.com/log/linux/maintainer-06.html>](http://www.kroah.com/log/linux/maintainer-06.html)

**NO!!!! No more huge patch bombs to linux-kernel@vger.kernel.org people!** <<https://lore.kernel.org/r/20050711.125305.08322243.davem@davemloft.net>>

**Kernel Documentation/process/coding-style.rst:** *Documentation/translations/zh\_CN/process/coding-style.rst*

**Linus Torvalds’ s mail on the canonical patch format:** <<https://lore.kernel.org/r/Pine.LNX.4.58.0504071023190.28951@ppc970.osdl.org>>

**Andi Kleen, “On submitting kernel patches”** Some strategies to get difficult or controversial changes in.

<http://halobates.de/on-submitting-patches.pdf>

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** <Documentation/process/programming-language.rst>

**Translator** Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

### 程序设计语言

内核是用 C 语言 *c-language* 编写的。更准确地说，内核通常是用 *gcc* 在 `-std=gnu11 gcc-c-dialect-options` 下编译的：ISO C11 的 GNU 方言

这种方言包含对语言 *gnu-extensions* 的许多扩展，当然，它们许多都在内核中使用。

对于一些体系结构，有一些使用 *clang* 和 *icc* 编译内核的支持，尽管在编写此文档时还没有完成，仍需要第三方补丁。

### 属性

在整个内核中使用的一个常见扩展是属性（attributes）*gcc-attribute-syntax* 属性允许将实现定义的语义引入语言实体（如变量、函数或类型），而无需对语言进行重大的语法更改（例如添加新关键字）[n2049](#)

在某些情况下，属性是可选的（即不支持这些属性的编译器仍然应该生成正确的代码，即使其速度较慢或执行的编译时检查/诊断次数不够）

内核定义了伪关键字（例如，`pure`），而不是直接使用 GNU 属性语法（例如，`_attribute_((__pure__))`），以检测可以使用哪些关键字和/或缩短代码，具体请参阅 `include/linux/compiler_attributes.h`

**c-language** <http://www.open-std.org/jtc1/sc22/wg14/www/standards>

**gcc** <https://gcc.gnu.org>

**clang** <https://clang.llvm.org>

**icc** <https://software.intel.com/en-us/c-compilers>

**c-dialect-options** <https://gcc.gnu.org/onlinedocs/gcc/C-Dialect-Options.html>

**gnu-extensions** <https://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html>

**gcc-attribute-syntax** <https://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html>

**n2049** <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2049.pdf>

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:<alexs@kernel.org>)。

---

**Original** Documentation/process/coding-style.rst

译者：

中文版维护者: 张乐 Zhang Le <r0bertz@gentoo.org>  
 中文版翻译者: 张乐 Zhang Le <r0bertz@gentoo.org>  
 中文版校译者: 王聪 Wang Cong <xifyou.wangcong@gmail.com>  
     wheelz <kernel.zeng@gmail.com>  
     管旭东 Xudong Guan <xudong.guan@gmail.com>  
     Li Zefan <lizf@cn.fujitsu.com>  
     Wang Chen <wangchen@cn.fujitsu.com>

## Linux 内核代码风格

这是一个简短的文档，描述了 linux 内核的首选代码风格。代码风格是因人而异的，而且我不愿意把自己的观点强加给任何人，但这就像我去做任何事情都必须遵循的原则那样，我也希望在绝大多数事上保持这种的态度。请(在写代码时)至少考虑一下这里的代码风格。

首先，我建议你打印一份 GNU 代码规范，然后不要读。烧了它，这是一个具有重大象征性意义的动作。

不管怎样，现在我们开始：

### 1) 缩进

制表符是 8 个字符，所以缩进也是 8 个字符。有些异端运动试图将缩进变为 4(甚至 2!) 字符深，这几乎相当于尝试将圆周率的值定义为 3。

理由：缩进的全部意义就在于清楚的定义一个控制块起止于何处。尤其是当你盯着你的屏幕连续看了 20 小时之后，你将会发现大一点的缩进会使你更容易分辨缩进。

现在，有些人会抱怨 8 个字符的缩进会使代码向右边移动的太远，在 80 个字符的终端屏幕上就很难读这样的代码。这个问题的答案是，如果你需要 3 级以上的缩进，不管用何种方式你的代码已经有问题了，应该修正你的程序。

简而言之，8 个字符的缩进可以让代码更容易阅读，还有一个好处是当你的函数嵌套太深的时候可以给你警告。留心这个警告。

在 switch 语句中消除多级缩进的首选的方式是让 switch 和从属于它的 case 标签对齐于同一列，而不要两次缩进 case 标签。比如：

```
switch (suffix) {
case 'G':
case 'g':
    mem <= 30;
    break;
case 'M':
case 'm':
    mem <= 20;
    break;
case 'K':
case 'k':
    mem <= 10;
```

```
    fallthrough;  
default:  
    break;  
}
```

不要把多个语句放在一行里，除非你有什么东西要隐藏：

```
if (condition) do_this;  
    do_something_evertime;
```

也不要一行里放多个赋值语句。内核代码风格超级简单。就是避免可能导致别人误读的表达式。

除了注释、文档和 Kconfig 之外，不要使用空格来缩进，前面的例子是例外，是有意为之。

选用一个好的编辑器，不要在行尾留空格。

## 2) 把长的行和字符串打散

代码风格的意义就在于使用平常使用的工具来维持代码的可读性和可维护性。

每一行的长度的限制是 80 列，我们强烈建议您遵守这个惯例。

长于 80 列的语句要打散成有意义的片段。除非超过 80 列能显著增加可读性，并且不会隐藏信息。子片段要明显短于母片段，并明显靠右。这同样适用于有着很长参数列表的函数头。然而，绝对不要打散对用户可见的字符串，例如 printk 信息，因为这样就很难对它们 grep。

## 3) 大括号和空格的放置

C 语言风格中另外一个常见问题是大括号的放置。和缩进大小不同，选择或弃用某种放置策略并没有多少技术上的原因，不过首选的方式，就像 Kernighan 和 Ritchie 展示给我们的，是把起始大括号放在行尾，而把结束大括号放在行首，所以：

```
if (x is true) {  
    we do y  
}
```

这适用于所有的非函数语句块 (if, switch, for, while, do)。比如：

```
switch (action) {  
case KOBJ_ADD:  
    return "add";  
case KOBJ_REMOVE:  
    return "remove";  
case KOBJ_CHANGE:  
    return "change";  
default:  
    return NULL;  
}
```

不过，有一个例外，那就是函数：函数的起始大括号放置于下一行的开头，所以：

```
int function(int x)
{
    body of function
}
```

全世界的异端可能会抱怨这个不一致性是…呃…不一致的，不过所有思维健全的人都知道 (a) K&R 是 **正确的** 并且 (b) K&R 是正确的。此外，不管怎样函数都是特殊的 (C 函数是不能嵌套的)。

注意结束大括号独自占据一行，除非它后面跟着同一个语句的剩余部分，也就是 do 语句中的“while”或者 if 语句中的“else”，像这样：

```
do {
    body of do-loop
} while (condition);
```

和

```
if (x == y) {
    ...
} else if (x > y) {
    ...
} else {
    ...
}
```

理由：K&R。

也请注意这种大括号的放置方式也能使空 (或者差不多空的) 行的数量最小化，同时不失可读性。因此，由于你的屏幕上的新行是不可再生资源 (想想 25 行的终端屏幕)，你将会有更多的空行来放置注释。

当只有一个单独的语句的时候，不用加不必要的大括号。

```
if (condition)
    action();
```

和

```
if (condition)
    do_this();
else
    do_that();
```

这并不适用于只有一个条件分支是单语句的情况；这时所有分支都要使用大括号：

```
if (condition) {
    do_this();
    do_that();
```

```
} else {
    otherwise();
}
```

### 3.1) 空格

Linux 内核的空格使用方式 (主要) 取决于它是用于函数还是关键字。(大多数) 关键字后要加一个空格。值得注意的例外是 `sizeof`, `typeof`, `alignof` 和 `_attribute_`, 这些关键字某些程度上看起来更像函数(它们在 Linux 里也常常伴随小括号而使用, 尽管在 C 里这样的小括号不是必需的, 就像 `struct fileinfo info;` 声明过后的 `sizeof info`)。

所以在这些关键字之后放一个空格:

```
if, switch, case, for, do, while
```

但是不要在 `sizeof`, `typeof`, `alignof` 或者 `_attribute_` 这些关键字之后放空格。例如,

```
s = sizeof(struct file);
```

不要在小括号里的表达式两侧加空格。这是一个 **反例**:

```
s = sizeof( struct file );
```

当声明指针类型或者返回指针类型的函数时, `*` 的首选使用方式是使之靠近变量名或者函数名, 而不是靠近类型名。例子:

```
char *linux_banner;
unsigned long long memparse(char *ptr, char **retptr);
char *match_strdup(substring_t *s);
```

在大多数二元和三元操作符两侧使用一个空格, 例如下面所有这些操作符:

```
= + - < > * / % | & ^ <= >= == != ? :
```

但是一元操作符后不要加空格:

```
& * + - ~ ! sizeof typeof alignof __attribute__ defined
```

后缀自加和自减一元操作符前不加空格:

```
++ --
```

前缀自加和自减一元操作符后不加空格:

```
++ --
```

. 和 -> 结构体成员操作符前后不加空格。

不要在行尾留空白。有些可以自动缩进的编辑器会在新行的行首加入适量的空白，然后你就可以直接在那一行输入代码。不过假如你最后没有在那一行输入代码，有些编辑器就不会移除已经加入的空白，就像你故意留下一个只有空白的行。包含行尾空白的行就这样产生了。

当 git 发现补丁包含了行尾空白的时候会警告你，并且可以应你的要求去掉行尾空白；不过如果你是正在打一系列补丁，这样做会导致后面的补丁失败，因为你改变了补丁的上下文。

## 4) 命名

C 是一个简朴的语言，你的命名也应该这样。和 Modula-2 和 Pascal 程序员不同，C 程序员不使用类似 ThisVariableIsATemporaryCounter 这样华丽的名字。C 程程序员会称那个变量为 tmp，这样写起来会更容易，而且至少不会令其难于理解。

不过，虽然混用大小写的名字是不提倡使用的，但是全局变量还是需要一个具描述性的名字。称一个全局函数为 foo 是一个难以饶恕的错误。

全局变量（只有当你真正需要它们的时候再用它）需要有一个具描述性的名字，就像全局函数。如果你有一个可以计算活动用户数量的函数，你应该叫它 count\_active\_users() 或者类似的名字，你不应该叫它 cntuser()。

在函数名中包含函数类型（所谓的匈牙利命名法）是脑子出了问题——编译器知道那些类型而且能够检查那些类型，这样做只能把程序员弄糊涂了。

本地变量名应该简短，而且能够表达相关的含义。如果你有一些随机的整数型的循环计数器，它应该被称为 i。叫它 loop\_counter 并无益处，如果它没有被误解的可能的话。类似的，tmp 可以用来称呼任意类型的临时变量。

如果你怕混淆了你的本地变量名，你就遇到另一个问题了，叫做函数增长荷尔蒙失衡综合症。请看第六章（函数）。

## 5) Typedef

不要使用类似 vps\_t 之类的东西。

对结构体和指针使用 typedef 是一个错误。当你在代码里看到：

```
vps_t a;
```

这代表什么意思呢？

相反，如果是这样

```
struct virtual_container *a;
```

你就知道 a 是什么了。

很多人认为 typedef 能提高可读性。实际不是这样的。它们只在下列情况下有用：

(a) 完全不透明的对象 (这种情况下要主动使用 `typedef` 来 隐藏这个对象实际上是什么)。

例如: `pte_t` 等不透明对象, 你只能用合适的访问函数来访问它们。

---

**Note:** 不透明性和“访问函数”本身是不好的。我们使用 `pte_t` 等类型的原因在于真的是完全没有任何共用的可访问信息。

---

(b) 清楚的整数类型, 如此, 这层抽象就可以 帮助消除到底是 `int` 还是 `long` 的混淆。

`u8/u16/u32` 是完全没有问题的 `typedef`, 不过它们更符合类别 (d) 而不是这里。

---

**Note:** 要这样做, 必须事出有因。如果某个变量是 `unsigned long`, 那么没有必要

```
typedef unsigned long myflags_t;
```

---

不过如果有一个明确的原因, 比如它在某种情况下可能会是一个 `unsigned int` 而在其他情况下可能为 `unsigned long`, 那么就不要犹豫, 请务必使用 `typedef`。

(c) 当你使用 `sparse` 按字面的创建一个 新类型来做类型检查的时候。

(d) 和标准 C99 类型相同的类型, 在某些例外的情况下。

虽然让眼睛和脑筋来适应新的标准类型比如 `uint32_t` 不需要花很多时间, 可是有些人仍然拒绝使用它们。

因此, Linux 特有的等同于标准类型的 `u8/u16/u32/u64` 类型和它们的有符号类型是被允许的——尽管在你自己的新代码中, 它们不是强制要求要使用的。

当编辑已经使用了某个类型集的已有代码时, 你应该遵循那些代码中已经做出的选择。

(e) 可以在用户空间安全使用的类型。

在某些用户空间可见的结构体里, 我们不能要求 C99 类型而且不能用上面提到的 `u32` 类型。因此, 我们在与用户空间共享的所有结构体中使用 `_u32` 和类似的类型。

可能还有其他的情况, 不过基本的规则是 永远不要使用 `typedef`, 除非你可以明确的应用上述某个规则中的一个。

总的来说, 如果一个指针或者一个结构体里的元素可以合理的被直接访问到, 那么它们就不应该是一个 `typedef`。

## 6) 函数

函数应该简短而漂亮，并且只完成一件事情。函数应该可以一屏或者两屏显示完（我们都应该知道 ISO/ANSI 屏幕大小是 80x24），只做一件事情，而且把它做好。

一个函数的最大长度是和该函数的复杂度和缩进级数成反比的。所以，如果你有一个理论上很简单的只有一个很长（但是简单）的 case 语句的函数，而且你需要在每个 case 里做很多很小的事情，这样的函数尽管很长，但也是可以的。

不过，如果你有一个复杂的函数，而且你怀疑一个天分不是很高的高中一年级学生可能甚至搞不清楚这个函数的目的，你应该严格遵守前面提到的长度限制。使用辅助函数，并为之取个具描述性的名字（如果你觉得它们的性能很重要的话，可以让编译器内联它们，这样的效果往往比你写一个复杂函数的效果要好。）

函数的另外一个衡量标准是本地变量的数量。此数量不应超过 5 – 10 个，否则你的函数就有问题了。重新考虑一下你的函数，把它分拆成更小的函数。人的大脑一般可以轻松的同时跟踪 7 个不同的事物，如果再增多的话，就会糊涂了。即便你聪颖过人，你也可能会记不清你 2 个星期前做过的事情。

在源文件里，使用空行隔开不同的函数。如果该函数需要被导出，它的 **EXPORT** 宏应该紧贴在它的结束大括号之下。比如：

```
int system_is_up(void)
{
    return system_state == SYSTEM_RUNNING;
}
EXPORT_SYMBOL(system_is_up);
```

在函数原型中，包含函数名和它们的数据类型。虽然 C 语言里没有这样的要求，在 Linux 里这是提倡的做法，因为这样可以很简单的给读者提供更多的有价值的信息。

## 7) 集中的函数退出途径

虽然被某些人声称已经过时，但是 goto 语句的等价物还是经常被编译器所使用，具体形式是无条件跳转指令。

当一个函数从多个位置退出，并且需要做一些类似清理的常见操作时，goto 语句就很方便了。如果并不需要清理操作，那么直接 return 即可。

选择一个能够说明 goto 行为或它为何存在的标签名。如果 goto 要释放 buffer，一个不错的名称可以是 `out_free_buffer:`。别去使用像 `err1:` 和 `err2:` 这样的 GW\_BASIC 名称，因为一旦你添加或删除了（函数的）退出路径，你就必须对它们重新编号，这样会难以去检验正确性。

使用 goto 的理由是：

- 无条件语句容易理解和跟踪
- 嵌套程度减小
- 可以避免由于修改时忘记更新个别的退出点而导致错误
- 让编译器省去删除冗余代码的工作；）

```
int fun(int a)
{
    int result = 0;
    char *buffer;

    buffer = kmalloc(SIZE, GFP_KERNEL);
    if (!buffer)
        return -ENOMEM;

    if (condition1) {
        while (loop1) {
            ...
        }
        result = 1;
        goto out_free_buffer;
    }
    ...
out_free_buffer:
    kfree(buffer);
    return result;
}
```

一个需要注意的常见错误是一个 err 错误，就像这样：

```
err:
    kfree(foo->bar);
    kfree(foo);
    return ret;
```

这段代码的错误是，在某些退出路径上 foo 是 NULL。通常情况下，通过把它分离成两个错误标签 `err_free_bar:` 和 `err_free_foo:` 来修复这个错误：

```
err_free_bar:
    kfree(foo->bar);
err_free_foo:
    kfree(foo);
    return ret;
```

理想情况下，你应该模拟错误来测试所有退出路径。

## 8) 注释

注释是好的，不过有过度注释的危险。永远不要在注释里解释你的代码是如何运作的：更好的做法是让别人一看你的代码就可以明白，解释写的很差的代码是浪费时间。

一般的，你想要你的注释告诉别人你的代码做了什么，而不是怎么做的。也请你不要把注释放在一个函数体内部：如果函数复杂到你需要独立的注释其中的一部分，你很可能需要回到第六章看一看。你可以做一些小注释来注明或警告某些很聪明（或者糟糕）的做法，但不要加太多。你应该做的，是把注释放在函数的头部，告诉人们它做了什么，也可以加上它做这些事情的原因。

当注释内核 API 函数时，请使用 kernel-doc 格式。请看 Documentation/doc-guide/ 和 scripts/kernel-doc 以获得详细信息。

长（多行）注释的首选风格是：

```
/*
 * This is the preferred style for multi-line
 * comments in the Linux kernel source code.
 * Please use it consistently.
 *
 * Description: A column of asterisks on the left side,
 * with beginning and ending almost-blank lines.
 */
```

对于在 net/ 和 drivers/net/ 的文件，首选的长（多行）注释风格有些不同。

```
/* The preferred comment style for files in net/ and drivers/net
 * looks like this.
 *
 * It is nearly the same as the generally preferred comment style,
 * but there is no initial almost-blank line.
 */
```

注释数据也是很重要的，不管是基本类型还是衍生类型。为了方便实现这一点，每一行应只声明一个数据（不要使用逗号来一次声明多个数据）。这样你就有空间来为每个数据写一段小注释来解释它们的用途了。

## 9) 你已经把事情弄糟了

这没什么，我们都是这样。可能你的使用了很长时间 Unix 的朋友已经告诉你 GNU emacs 能自动帮你格式化 C 源代码，而且你也注意到了，确实是这样，不过它所使用的默认值和我们想要的相去甚远（实际上，甚至比随机打的还要差——无数个猴子在 GNU emacs 里打字永远不会创造出一个好程序）（译注：Infinite Monkey Theorem）

所以你要么放弃 GNU emacs，要么改变它让它使用更合理的设定。要采用后一个方案，你可以把下面这段粘贴到你的.emacs 文件里。

```
(defun c-lineup-arglist-tabs-only (ignored)
  "Line up argument lists by tabs, not spaces"
  (let* ((anchor (c-lang-elem-pos c-syntactic-element))
         (column (c-lang-elem-2nd-pos c-syntactic-element))
         (offset (- (1+ column) anchor))
         (steps (floor offset c-basic-offset)))
    (* (max steps 1)
       c-basic-offset)))

(dir-locals-set-class-variables
 'linux-kernel
 '((c-mode . (
   (c-basic-offset . 8)
   (c-label-minimum-indentation . 0)
   (c-offsets-alist . (
     (arglist-close . c-lineup-arglist-tabs-only)
     (arglist-cont-nonempty .
      (c-lineup-gcc-asm-reg c-lineup-arglist-tabs-only))
     (arglist-intro . +)
     (brace-list-intro . +)
     (c . c-lineup-C-comments)
     (case-label . 0)
     (comment-intro . c-lineup-comment)
     (cpp-define-intro . +)
     (cpp-macro . -1000)
     (cpp-macro-cont . +)
     (defun-block-intro . +)
     (else-clause . 0)
     (func-decl-cont . +)
     (inclass . +)
     (inher-cont . c-lineup-multi-inher)
     (knr-argdecl-intro . 0)
     (label . -1000)
     (statement . 0)
     (statement-block-intro . +)
     (statement-case-intro . +)
     (statement-cont . +)
     (substatement . +)
     )))
   (indent-tabs-mode . t)
   (show-trailing-whitespace . t)
 ))))

(dir-locals-set-directory-class
 (expand-file-name "~/src/linux-trees")
 'linux-kernel)
```

这会让 emacs 在 ~/src/linux-trees 下的 C 源文件获得更好的内核代码风格。

不过就算你尝试让 emacs 正确的格式化代码失败了，也并不意味着你失去了一切：还可以用 indent。

不过，GNU indent 也有和 GNU emacs 一样有问题的设定，所以你需要给它一些命令选项。不过，这还不算太糟糕，因为就算是 GNU indent 的作者也认同 K&R 的权威性 (GNU 的人并不是坏人，他们只是在这个问题上被严重的误导了)，所以你只要给 indent 指定选项 `-kr -i8` (代表 K&R，8 字符缩进)，或使用 `scripts/Lindent` 这样就可以以最时髦的方式缩进源代码。

`indent` 有很多选项，特别是重新格式化注释的时候，你可能需要看一下它的手册。不过记住：`indent` 不能修正坏的编程习惯。

## 10) Kconfig 配置文件

对于遍布源码树的所有 Kconfig\* 配置文件来说，它们缩进方式有所不同。紧挨着 `config` 定义的行，用一个制表符缩进，然而 `help` 信息的缩进则额外增加 2 个空格。举个例子：

```
config AUDIT
    bool "Auditing support"
    depends on NET
    help
        Enable auditing infrastructure that can be used with another
        kernel subsystem, such as SELinux (which requires this for
        logging of avc messages output). Does not do system-call
        auditing without CONFIG_AUDITSYSCALL.
```

而那些危险的功能 (比如某些文件系统的写支持) 应该在它们的提示字符串里显著的声明这一点：

```
config ADFS_FS_RW
    bool "ADFS write support (DANGEROUS)"
    depends on ADFS_FS
    ...
```

要查看配置文件的完整文档，请看 [Documentation/kbuild/kconfig-language.rst](#)。

## 11) 数据结构

如果一个数据结构，在创建和销毁它的单线执行环境之外可见，那么它必须要有一个引用计数器。内核里没有垃圾收集 (并且内核之外的垃圾收集慢且效率低下)，这意味着你绝对需要记录你对这种数据结构的使用情况。

引用计数意味着你能够避免上锁，并且允许多个用户并行访问这个数据结构——而不需要担心这个数据结构仅仅因为暂时不被使用就消失了，那些用户可能不过是沉睡了一阵或者做了一些其他事情而已。

注意上锁 **不能** 取代引用计数。上锁是为了保持数据结构的一致性，而引用计数是一个内存管理技巧。通常二者都需要，不要把两个搞混了。

很多数据结构实际上有 2 级引用计数，它们通常有不同类的用户。子类计数器统计子类用户的数量，每当子类计数器减至零时，全局计数器减一。

这种 多级引用计数的例子可以在内存管理 (`struct mm_struct: mm_users` 和 `mm_count`)，和文件系统 (`struct super_block: s_count` 和 `s_active`) 中找到。

记住：如果另一个执行线索可以找到你的数据结构，但这个数据结构没有引用计数器，这里几乎肯定是一个 bug。

## 12) 宏, 枚举和 RTL

用于定义常量的宏的名字及枚举里的标签需要大写。

```
#define CONSTANT 0x12345
```

在定义几个相关的常量时，最好用枚举。

宏的名字请用大写字母，不过形如函数的宏的名字可以用小写字母。

一般的，如果能写成内联函数就不要写成像函数的宏。

含有多个语句的宏应该被包含在一个 do-while 代码块里：

```
#define macrofun(a, b, c)
    do {
        if (a == 5)
            do_this(b, c);
    } while (0)
```

使用宏的时候应避免的事情：

1) 影响控制流程的宏：

```
#define FOO(x)
    do {
        if (blah(x) < 0)
            return -EBUGGERED;
    } while (0)
```

非常不好。它看起来像一个函数，不过却能导致调用它的函数退出；不要打乱读者大脑里的语法分析器。

2) 依赖于一个固定名字的本地变量的宏：

```
#define FOO(val) bar(index, val)
```

可能看起来像是个不错的东西，不过它非常容易把读代码的人搞糊涂，而且容易导致看起来不相关的改动带来错误。

3) 作为左值的带参数的宏：FOO(x) = y；如果有人把 FOO 变成一个内联函数的话，这种用法就会出错了。

4) 忘记了优先级：使用表达式定义常量的宏必须将表达式置于一对小括号之内。带参数的宏也要注意此类问题。

```
#define CONSTANT 0x4000
#define CONSTEXP (CONSTANT / 3)
```

5) 在宏里定义类似函数的本地变量时命名冲突:

```
#define FOO(x)
({
    typeof(x) ret;
    ret = calc_ret(x);
    (ret);
})
```

ret 是本地变量的通用名字 - \_\_foo\_ret 更不容易与一个已存在的变量冲突。

cpp 手册对宏的讲解很详细。gcc internals 手册也详细讲解了 RTL，内核里的汇编语言经常用到它。

## 13) 打印内核消息

内核开发者应该是受过良好教育的。请一定注意内核信息的拼写，以给人以好的印象。不要用不规范的单词比如 `dont`，而要用 `do not` 或者 `don't`。保证这些信息简单明了，无歧义。

内核信息不必以英文句号结束。

在小括号里打印数字 (%d) 没有任何价值，应该避免这样做。

<linux/device.h> 里有一些驱动模型诊断宏，你应该使用它们，以确保信息对应于正确的设备和驱动，并且被标记了正确的消息级别。这些宏有: `dev_err()`, `dev_warn()`, `dev_info()` 等等。对于那些不和某个特定设备相关连的信息，<linux/printk.h> 定义了 `pr_notice()`, `pr_info()`, `pr_warn()`, `pr_err()` 和其他。

写出好的调试信息可以是一个很大的挑战；一旦你写出后，这些信息在远程除错时能提供极大的帮助。然而打印调试信息的处理方式同打印非调试信息不同。其他 `pr_XXX()` 函数能无条件地打印，`pr_debug()` 却不；默认情况下它不会被编译，除非定义了 `DEBUG` 或设定了 `CONFIG_DYNAMIC_DEBUG`。实际这同样是为了 `dev_dbg()`，一个相关约定是在一个已经开启了 `DEBUG` 时，使用 `VERBOSE_DEBUG` 来添加 `dev_vdbg()`。

许多子系统拥有 Kconfig 调试选项来开启 `-DDEBUG` 在对应的 Makefile 里面；在其他情况下，特殊文件使用 `#define DEBUG`。当一条调试信息需要被无条件打印时，例如，如果已经包含一个调试相关的 `#ifdef` 条件，`printk(KERN_DEBUG ...)` 就可被使用。

## 14) 分配内存

内核提供了下面的一般用途的内存分配函数: `kmalloc()`, `kzalloc()`, `kmalloc_array()`, `kcalloc()`, `vmalloc()` 和 `vzalloc()`。请参考 API 文档以获取有关它们的详细信息。

传递结构体大小的首选形式是这样的：

```
p = kmalloc(sizeof(*p), ...);
```

另外一种传递方式中，`sizeof` 的操作数是结构体的名字，这样会降低可读性，并且可能会引入 bug。有可能指针变量类型被改变时，而对应的传递给内存分配函数的 `sizeof` 的结果不变。

强制转换一个 void 指针返回值是多余的。C 语言本身保证了从 void 指针到其他任何指针类型的转换是没有问题的。

分配一个数组的首选形式是这样的：

```
p = kmalloc_array(n, sizeof(...), ...);
```

分配一个零长数组的首选形式是这样的：

```
p = kcalloc(n, sizeof(...), ...);
```

两种形式检查分配大小  $n * \text{sizeof}(\cdots)$  的溢出，如果溢出返回 NULL。

## 15) 内联弊病

有一个常见的误解是 内联是 gcc 提供的可以让代码运行更快的一个选项。虽然使用内联函数有时候是恰当的（比如作为一种替代宏的方式，请看第十二章），不过很多情况下不是这样。inline 的过度使用会使内核变大，从而使整个系统运行速度变慢。因为体积大内核会占用更多的指令高速缓存，而且会导致 pagecache 的可用内存减少。想象一下，一次 pagecache 未命中就会导致一次磁盘寻址，将耗时 5 毫秒。5 毫秒的时间内 CPU 能执行很多很多指令。

一个基本的原则是如果一个函数有 3 行以上，就不要把它变成内联函数。这个原则的一个例外是，如果你知道某个参数是一个编译时常量，而且因为这个常量你确定编译器在编译时能优化掉你的函数的大部分代码，那仍然可以给它加上 inline 关键字。kmalloc() 内联函数就是一个很好的例子。

人们经常主张给 static 的而且只用了一次的函数加上 inline，如此不会有任何损失，因为没有什么好权衡的。虽然从技术上说这是正确的，但是实际上这种情况下即使不加 inline gcc 也可以自动使其内联。而且其他用户可能会要求移除 inline，由此而来的争论会抵消 inline 自身的潜在价值，得不偿失。

## 16) 函数返回值及命名

函数可以返回多种不同类型的值，最常见的一种是表明函数执行成功或者失败的值。这样的一个值可以表示为一个错误代码整数 (-Exxx = 失败, 0 = 成功) 或者一个 成功布尔值 (0 = 失败, 非 0 = 成功)。

混合使用这两种表达方式是难于发现的 bug 的来源。如果 C 语言本身严格区分整形和布尔型变量，那么编译器就能够帮我们发现这些错误…不过 C 语言不区分。为了避免产生这种 bug，请遵循下面的惯例：

如果函数的名字是一个动作或者强制性的命令，那么这个函数应该返回错误代码整数。如果是一个判断，那么函数应该返回一个 “成功” 布尔值。

比如，add\_work 是一个命令，所以 add\_work() 在成功时返回 0，在失败时返回 -EBUSY。类似的，因为 PCI device present 是一个判断，所以 pci\_dev\_present() 在成功找到一个匹配的设备时应该返回 1，如果找不到时应该返回 0。

所有 EXPORTed 函数都必须遵守这个惯例，所有的公共函数也都应该如此。私有 (static) 函数不需要如此，但是我们也推荐这样做。

返回值是实际计算结果而不是计算是否成功的标志的函数不受此惯例的限制。一般的，他们通过返回一些正常值范围之外的结果来表示出错。典型的例子是返回指针的函数，他们使用 NULL 或者 ERR\_PTR 机制来报告错误。

## 17) 不要重新发明内核宏

头文件 include/linux/kernel.h 包含了一些宏，你应该使用它们，而不要自己写一些它们的变种。比如，如果你需要计算一个数组的长度，使用这个宏

```
#define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))
```

类似的，如果你要计算某结构体成员的大小，使用

```
#define sizeof_field(t, f) (sizeof(((t*)0)->f))
```

还有可以做严格的类型检查的 min() 和 max() 宏，如果你需要可以使用它们。你可以自己看看那个头文件里还定义了什么你可以拿来用的东西，如果有定义的话，你就不应在你的代码里自己重新定义。

## 18) 编辑器模式行和其他需要罗嗦的事情

有一些编辑器可以解释嵌入在源文件里的由一些特殊标记标明的配置信息。比如，emacs 能够解释被标记成这样的行：

```
-*- mode: c -*-
```

或者这样的：

```
/*
Local Variables:
compile-command: "gcc -DMAGIC_DEBUG_FLAG foo.c"
End:
*/
```

Vim 能够解释这样的标记：

```
/* vim:set sw=8 noet */
```

不要在源代码中包含任何这样的内容。每个人都有他自己的编辑器配置，你的源文件不应该覆盖别人的配置。这包括有关缩进和模式配置的标记。人们可以使用他们自己定制的模式，或者使用其他可以产生正确的缩进的巧妙方法。

## 19) 内联汇编

在特定架构的代码中，你可能需要内联汇编与 CPU 和平台相关功能连接。需要这么做时就不要犹豫。然而，当 C 可以完成工作时，不要平白无故地使用内联汇编。在可能的情况下，你可以并且应该用 C 和硬件沟通。

请考虑去写捆绑通用位元 (wrap common bits) 的内联汇编的简单辅助函数，别去重复地写下只有细微差异内联汇编。记住内联汇编可以使用 C 参数。

大型，有一定复杂度的汇编函数应该放在.S 文件内，用相应的 C 原型定义在 C 头文件中。汇编函数的 C 原型应该使用 `asm linkage`。

你可能需要把汇编语句标记为 `volatile`，用来阻止 GCC 在没发现任何副作用后就把它移除了。你不必总是这样做，尽管，这不必要的举动会限制优化。

在写一个包含多条指令的单个内联汇编语句时，把每条指令用引号分割而且各占一行，除了最后一条指令外，在每个指令结尾加上 `nt`，让汇编输出时可以正确地缩进下一条指令：

```
asm ("magic %reg1, #42\n\t"
     "more_magic %reg2, %reg3"
     : /* outputs */ : /* inputs */ : /* clobbers */);
```

## 20) 条件编译

只要可能，就不要在.c 文件里面使用预处理条件 (`#if`, `#ifdef`)；这样做让代码更难阅读并且更难去跟踪逻辑。替代方案是，在头文件中用预处理条件提供给那些.c 文件使用，再给 `#else` 提供一个空桩 (no-op stub) 版本，然后在.c 文件内无条件地调用那些 (定义在头文件内的) 函数。这样做，编译器会避免为桩函数 (stub) 的调用生成任何代码，产生的结果是相同的，但逻辑将更加清晰。

最好倾向于编译整个函数，而不是函数的一部分或表达式的一部分。与其放一个 `ifdef` 在表达式内，不如分解出部分或全部表达式，放进一个单独的辅助函数，并应用预处理条件到这个辅助函数内。

如果你有一个在特定配置中，可能变成未使用的函数或变量，编译器会警告它定义了但未使用，把它标记为 `_maybe_unused` 而不是将它包含在一个预处理条件中。(然而，如果一个函数或变量总是未使用，就直接删除它。)

在代码中，尽可能地使用 `IS_ENABLED` 宏来转化某个 Kconfig 标记为 C 的布尔表达式，并在一般的 C 条件中使用它：

```
if (IS_ENABLED(CONFIG_SOMETHING)) {
    ...
}
```

编译器会做常量折叠，然后就像使用 `#ifdef` 那样去包含或排除代码块，所以这不会带来任何运行时开销。然而，这种方法依旧允许 C 编译器查看块内的代码，并检查它的正确性 (语法，类型，符号引用，等等)。因此，如果条件不满足，代码块内的引用符号就不存在时，你还是必须去用 `#ifdef`。

在任何有意义的 `#if` 或 `#ifdef` 块的末尾 (超过几行的)，在 `#endif` 同一行的后面写下注解，注释这个条件表达式。例如：

```
#ifdef CONFIG_SOMETHING
...
#endif /* CONFIG_SOMETHING */
```

## 附录 I) 参考

The C Programming Language, 第二版作者: Brian W. Kernighan 和 Denni M. Ritchie. Prentice Hall, Inc., 1988. ISBN 0-13-110362-8 (软皮), 0-13-110370-9 (硬皮).

The Practice of Programming 作者: Brian W. Kernighan 和 Rob Pike. Addison-Wesley, Inc., 1999. ISBN 0-201-61586-X.

GNU 手册 - 遵循 K&R 标准和此文本 - cpp, gcc, gcc internals and indent, 都可以从 <https://www.gnu.org/manual/> 找到

WG14 是 C 语言的国际标准化工作组, URL: <http://www.open-std.org/JTC1/SC22/WG14/>

Kernel process/coding-style.rst, 作者 greg@kroah.com 发表于 OLS 2002: [http://www.kroah.com/linux/talks/ols\\_2002\\_kernel\\_codingstyle\\_talk/html/](http://www.kroah.com/linux/talks/ols_2002_kernel_codingstyle_talk/html/)

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解, 而不是作为一个分支。因此, 如果您对此文件有任何意见或更新, 请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题, 请联系该文件的译者, 或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/process/development-process.rst

**Translator** Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

## 内核开发过程指南

内容:

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解, 而不是作为一个分支。因此, 如果您对此文件有任何意见或更新, 请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题, 请联系该文件的译者, 或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/process/1.Intro.rst

**Translator** 时奎亮 Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

校译 吴想成 Wu XiangCheng <[bowxc@email.cn](mailto:bowxc@email.cn)>

## 引言

### 内容提要

本节的其余部分涵盖了内核开发的过程，以及开发人员及其雇主在这方面可能遇到的各种问题。有很多原因使内核代码应被合并到正式的（“主线”）内核中，包括对用户的自动可用性、多种形式的社区支持以及影响内核开发方向的能力。提供给 Linux 内核的代码必须在与 GPL 兼容的许可证下可用。

[开发流程如何进行](#) 介绍了开发过程、内核发布周期和合并窗口的机制。涵盖了补丁开发、审查和合并周期中的各个阶段。还有一些关于工具和邮件列表的讨论？鼓励希望开始内核开发的开发人员跟踪并修复缺陷以作为初步练习。

[早期规划](#) 包括项目的早期规划，重点是尽快让开发社区参与进来。

[使代码正确](#) 是关于编程过程的；介绍了其他开发人员遇到的几个陷阱。也涵盖了对补丁的一些要求，并且介绍了一些工具，这些工具有助于确保内核补丁是正确的。

[发布补丁](#) 描述发布补丁以供评审的过程。为了让开发社区能认真对待，补丁必须被正确格式化和描述，并且必须发送到正确的地方。遵循本节中的建议有助于确保您的工作能被较好地接纳。

[跟进](#) 介绍了发布补丁之后发生的事情；工作在这时还远远没有完成。与审阅者一起工作是开发过程中一个重要部分；本节提供了一些关于如何在这个重要阶段避免问题的提示。当补丁被合并到主线中时，开发人员要注意不要假定任务已经完成。

[高级主题](#) 介绍了两个“高级”主题：使用 Git 管理补丁和查看其他人发布的补丁。

[更多信息](#) 总结了有关内核开发的更多信息，附带有相关资源链接。

### 这个文档是关于什么的

Linux 内核有超过 800 万行代码，每个版本的贡献者超过 1000 人，是现存最大、最活跃的免费软件项目之一。从 1991 年开始，这个内核已经发展成为一个最好的操作系统组件，运行在袖珍数字音乐播放器、台式电脑、现存最大的超级计算机以及所有类型的系统上。它是一种适用于几乎任何情况的健壮、高效和可扩展的解决方案。

随着 Linux 的发展，希望参与其开发的开发人员（和公司）的数量也在增加。硬件供应商希望确保 Linux 能够很好地支持他们的产品，使这些产品对 Linux 用户具有吸引力。嵌入式系统供应商使用 Linux 作为集成产品的组件，希望 Linux 能够尽可能地胜任手头的任务。分销商和其他基于 Linux 的软件供应商切实关心 Linux 内核的功能、性能和可靠性。最终用户也常常希望修改 Linux，使之能更好地满足他们的需求。

Linux 最引人注目的特性之一是这些开发人员可以访问它；任何具备必要技能的人都可以改进 Linux 并影响其发展方向。专有产品不能提供这种开放性，这是自由软件的一个特点。如果有什么不同的话，那就是内核

比大多数其他自由软件项目更开放。一个典型的三个月内核开发周期可以涉及 1000 多个开发人员，他们为 100 多个不同的公司（或者根本不隶属公司）工作。

与内核开发社区合作并不是特别困难。但尽管如此，仍有许多潜在的贡献者在尝试做内核工作时遇到了困难。内核社区已经发展出自己独特的操作方式，使其能够在每天都要更改数千行代码的环境中顺利运行（并生成高质量的产品）。因此，Linux 内核开发过程与专有的开发模式有很大的不同也就不足为奇了。

对于新开发人员来说，内核的开发过程可能会让人感到奇怪和恐惧，但这背后有充分的理由和坚实的经验。一个不了解内核社区工作方式的开发人员（或者更糟的是，他们试图抛弃或规避之）会得到令人沮丧的体验。开发社区在帮助那些试图学习的人的同时，没有时间帮助那些不愿意倾听或不关心开发过程的人。

希望阅读本文的人能够避免这种令人沮丧的经历。这些材料很长，但阅读它们时所做的努力会在短时间内得到回报。开发社区总是需要能让内核变更好的开发人员；下面的文字应该帮助您或为您工作的人员加入我们的社区。

## 致谢

本文档由 Jonathan Corbet <[corbet@lwn.net](mailto:corbet@lwn.net)> 撰写。以下人员的建议使之更为完善：Johannes Berg, James Berry, Alex Chiang, Roland Dreier, Randy Dunlap, Jake Edge, Jiri Kosina, Matt Mackall, Arthur Marsh, Amanda McPherson, Andrew Morton, Andrew Price, Tsugikazu Shibata 和 Jochen Voß。

这项工作得到了 Linux 基金会的支持，特别感谢 Amanda McPherson，他看到了这项工作的价值并将其变成现实。

## 代码进入主线的重要性

有些公司和开发人员偶尔会想，为什么他们要费心学习如何与内核社区合作，并将代码放入主线内核（“主线”是由 Linus Torvalds 维护的内核，Linux 发行商将其用作基础）。在短期内，贡献代码看起来像是一种可以避免的开销；维护独立代码并直接支持用户似乎更容易。事实上，保持代码独立（“树外”）是在经济上是错误的。

为了说明树外代码成本，下面给出内核开发过程的一些相关方面；本文稍后将更详细地讨论其中的大部分内容。请考虑：

- 所有 Linux 用户都可以使用合并到主线内核中的代码。它将自动出现在所有启用它的发行版上。无需驱动程序磁盘、额外下载，也不需要为多个发行版的多个版本提供支持；这一切将方便所有开发人员和用户。并入主线解决了大量的分发和支持问题。
- 当内核开发人员努力维护一个稳定的用户空间接口时，内核内部 API 处于不断变化之中。不维持稳定的内部接口是一个慎重的设计决策；它允许在任何时候进行基本的改进，并产出更高质量的代码。但该策略导致结果是，若要使用新的内核，任何树外代码都需要持续的维护。维护树外代码会需要大量的工作才能使代码保持正常运行。

相反，位于主线中的代码不需要这样做，因为基本规则要求进行 API 更改的任何开发人员也必须修复由于该更改而破坏的任何代码。因此，合并到主线中的代码大大降低了维护成本。

- 除此之外，内核中的代码通常会被其他开发人员改进。您授权的用户社区和客户对您产品的改进可能会令人惊喜。
- 内核代码在合并到主线之前和之后都要经过审查。无论原始开发人员的技能有多强，这个审查过程总是能找到改进代码的方法。审查经常发现严重的错误和安全问题。对于在封闭环境中开发的代码尤其如此；这种代码从外部开发人员的审查中获益匪浅。树外代码是低质量代码。
- 参与开发过程是您影响内核开发方向的方式。旁观者的抱怨会被听到，但是活跃的开发人员有更强的声音——并且能够实现使内核更好地满足其需求的更改。
- 当单独维护代码时，总是存在第三方为类似功能提供不同实现的可能性。如果发生这种情况，合并代码将变得更加困难——甚至成为不可能。之后，您将面临以下令人不快的选择：(1) 无限期地维护树外的非标准特性，或 (2) 放弃代码并将用户迁移到树内版本。
- 代码的贡献是使整个流程工作的根本。通过贡献代码，您可以向内核添加新功能，并提供其他内核开发人员使用的功能和示例。如果您已经为 Linux 开发了代码（或者正在考虑这样做），那么您显然对这个平台的持续成功感兴趣；贡献代码是确保成功的最好方法之一。

上述所有理由都适用于任何树外内核代码，包括以专有的、仅二进制形式分发的代码。然而，在考虑任何类型的纯二进制内核代码分布之前，还需要考虑其他因素。包括：

- 围绕专有内核模块分发的法律问题其实较为模糊；相当多的内核版权所有者认为，大多数仅二进制的模块是内核的派生产品，因此，它们的分发违反了 GNU 通用公共许可证（下面将详细介绍）。本文作者不是律师，本文档中的任何内容都不可能被视为法律建议。封闭源代码模块的真实法律地位只能由法院决定。但不管怎样，困扰这些模块的不确定性仍然存在。
- 二进制模块大大增加了调试内核问题的难度，以至于大多数内核开发人员甚至都不会尝试。因此，只分发二进制模块将使您的用户更难从社区获得支持。
- 对于仅二进制的模块的发行者来说，支持也更加困难，他们必须为他们希望支持的每个发行版和每个内核版本提供不同版本的模块。为了提供较为全面的覆盖范围，可能需要一个模块的几十个构建，并且每次升级内核时，您的用户都必须单独升级这些模块。
- 上面提到的关于代码评审的所有问题都更加存在于封闭源代码中。由于该代码根本不可得，因此社区无法对其进行审查，毫无疑问，它将存在严重问题。

尤其是嵌入式系统的制造商，可能会倾向于忽视本节中所说的大部分内容；因为他们相信自己正在商用一种使用冻结内核版本的独立产品，在发布后不需要再进行开发。这个论点忽略了广泛的代码审查的价值以及允许用户向产品添加功能的价值。但这些产品的商业寿命有限，之后必须发布新版本的产品。在这一点上，代码在主线上并得到良好维护的供应商将能够更好地占位，以使新产品快速上市。

## 许可

代码是根据一些许可证提供给 Linux 内核的，但是所有代码都必须与 GNU 通用公共许可证（GPLv2）的版本 2 兼容，该版本是覆盖整个内核分发的许可证。在实践中，这意味着所有代码贡献都由 GPLv2（可选地，语言允许在更高版本的 GPL 下分发）或 3 子句 BSD 许可（New BSD License, 译者注）覆盖。任何不包含在兼容许可证中的贡献都不会被接受到内核中。

贡献给内核的代码不需要（或请求）版权分配。合并到主线内核中的所有代码都保留其原始所有权；因此，内核现在拥有数千个所有者。

这种所有权结构也暗示着，任何改变内核许可的尝试都注定会失败。很少有实际情况可以获得所有版权所有者的同意（或者从内核中删除他们的代码）。因此，尤其是在可预见的将来，许可证不大可能迁移到 GPL 的版本 3。

所有贡献给内核的代码都必须是合法的免费软件。因此，不接受匿名（或化名）贡献者的代码。所有贡献者都需要在他们的代码上“sign off（签发）”，声明代码可以在 GPL 下与内核一起分发。无法提供未被其所有者许可为免费软件的代码，或可能为内核造成版权相关问题的代码（例如，由缺乏适当保护的反向工程工作派生的代码）不能被接受。

有关版权问题的提问在 Linux 开发邮件列表中很常见。这样的问题通常会得到不少答案，但请记住，回答这些问题的人不是律师，不能提供法律咨询。如果您有关于 Linux 源代码的法律问题，没有什么可以代替咨询了解这一领域的律师。依赖从技术邮件列表中获得的答案是一件冒险的事情。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

**Original** Documentation/process/2.Process.rst

**Translator** 时奎亮 Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

校译 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

## 开发流程如何进行

90 年代早期的 Linux 内核开发是一件相当松散的事情，涉及的用户和开发人员相对较少。由于拥有数以百万计的用户群，且每年有大约 2000 名开发人员参与进来，内核因此必须发展出许多既定流程来保证开发的顺利进行。要参与到流程中来，需要对此流程的进行方式有一个扎实的理解。

## 总览

内核开发人员使用一个松散的基于时间的发布过程，每两到三个月发布一次新的主要内核版本。最近的发布历史记录如下：

5.0	2019 年 3 月 3 日
5.1	2019 年 5 月 5 日
5.2	2019 年 7 月 7 日
5.3	2019 年 9 月 15 日
5.4	2019 年 11 月 24 日
5.5	2020 年 1 月 6 日

每个 5.x 版本都是一个主要的内核版本，具有新特性、内部 API 更改等等。一个典型的 5.x 版本包含大约 13000 个变更集，变更了几十万行代码。因此，5.x 是 Linux 内核开发的前沿；内核使用滚动开发模型，不断集成重大变化。

对于每个版本的补丁合并，遵循一个相对简单的规则。在每个开发周期的开头，“合并窗口”被打开。这时，被认为足够稳定（并且被开发社区接受）的代码被合并到主线内核中。在这段时间内，新开发周期的大部分变更（以及所有主要变更）将以接近每天 1000 次变更（“补丁”或“变更集”）的速度合并。

（顺便说一句，值得注意的是，合并窗口期间集成的更改并不是凭空产生的；它们是经提前收集、测试和分级的。稍后将详细描述该过程的工作方式。）

合并窗口持续大约两周。在这段时间结束时，Linus Torvalds 将声明窗口已关闭，并释放第一个“rc”内核。例如，对于目标为 5.6 的内核，在合并窗口结束时发生的释放将被称为 5.6-rc1。-rc1 版本是一个信号，表示合并新特性的时间已经过去，稳定下一个内核的时间已经到来。

在接下来的 6 到 10 周内，只有修复问题的补丁才应该提交给主线。有时会允许更大的更改，但这种情况很少发生；试图在合并窗口外合并新功能的开发人员往往受到友好的接待。一般来说，如果您错过了给定特性的合并窗口，最好的做法是等待下一个开发周期。（偶尔会对未支持硬件的驱动程序进行例外；如果它们不改变已有代码，则不会导致回归，应该可以随时被安全地加入）。

随着修复程序进入主线，补丁速度将随着时间的推移而变慢。Linus 大约每周发布一次新的-rc 内核；在内核被认为足够稳定并最终发布前，一般会达到-rc6 到-rc9 之间。然后，整个过程又重新开始了。

例如，这里是 5.4 的开发周期进行情况（2019 年）：

九月 15	5.3 稳定版发布
九月 30	5.4-rc1 合并窗口关闭
十月 6	5.4-rc2
十月 13	5.4-rc3
十月 20	5.4-rc4
十月 27	5.4-rc5
十一月 3	5.4-rc6
十一月 10	5.4-rc7
十一月 17	5.4-rc8
十一月 24	5.4 稳定版发布

开发人员如何决定何时结束开发周期并创建稳定版本？最重要的指标是以前版本的回归列表。不欢迎出现任何错误，但是那些破坏了以前能工作的系统的错误被认为是特别严重的。因此，导致回归的补丁是不受欢迎的，很可能在稳定期内删除。

开发人员的目标是在稳定发布之前修复所有已知的回归。在现实世界中，这种完美是很难实现的；在这种规模的项目中，变数太多了。需要说明的是，延迟最终版本只会使问题变得更糟；等待下一个合并窗口的更改将变多，导致下次出现更多的回归错误。因此，大多数 5.x 内核都有一些已知的回归错误，不过，希望没有一个是严重的。

一旦一个稳定的版本发布，它的持续维护工作就被移交给“稳定团队”，目前由 Greg Kroah-Hartman 领导。稳定团队将使用 5.x.y 编号方案不定期地发布稳定版本的更新。要合入更新版本，补丁必须（1）修复一个重要的缺陷，且（2）已经合并到下一个开发版本主线中。内核通常会在其初始版本后的一个以上的开发周期内收到稳定版更新。例如，5.2 内核的历史如下（2019 年）：

七月 7	5.2 稳定版发布
七月 13	5.2.1
七月 21	5.2.2
七月 26	5.2.3
七月 28	5.2.4
七月 31	5.2.5
...	...
十月 11	5.2.21

5.2.21 是 5.2 版本的最终稳定更新。

有些内核被指定为“长期”内核；它们将得到更长时间的支持。在本文中，当前的长期内核及其维护者是：

3.16	Ben Hutchings	(长期稳定内核)
4.4	Greg Kroah-Hartman & Sasha Levin	(长期稳定内核)
4.9	Greg Kroah-Hartman & Sasha Levin	
4.14	Greg Kroah-Hartman & Sasha Levin	
4.19	Greg Kroah-Hartman & Sasha Levin	
5.4	Greg Kroah-Hartman & Sasha Levin	

长期支持内核的选择纯粹是维护人员是否有需求和时间来维护该版本的问题。目前还没有为即将发布的任何特定版本提供长期支持的已知计划。

## 补丁的生命周期

补丁不会直接从开发人员的键盘进入主线内核。相反，有一个稍微复杂（如果有些非正式）的过程，旨在确保对每个补丁进行质量审查，并确保每个补丁实现了一个在主线中需要的更改。对于小的修复，这个过程可能会很快完成，而对于较大或有争议的变更，可能会持续数年。许多开发人员的沮丧来自于对这个过程缺乏理解或者试图绕过它。

为了减少这种挫败，本文将描述补丁如何进入内核。下面的介绍以一种较为理想化的方式描述了这个过程。更详细的过程将在后面的章节中介绍。

补丁通常要经历以下阶段：

- 设计。这就是补丁的真正需求——以及满足这些需求的方式——所在。设计工作通常是在不涉及社区的情况下完成的，但是如果可能的话，最好是在公开的情况下完成这项工作；这样可以节省很多稍后再重新设计的时间。
- 早期评审。补丁被发布到相关的邮件列表中，列表中的开发人员会回复他们可能有的任何评论。如果一切顺利的话，这个过程应该会发现补丁的任何主要问题。
- 更广泛的评审。当补丁接近准备好纳入主线时，它应该被相关的子系统维护人员接受——尽管这种接受并不能保证补丁会一直延伸到主线。补丁将出现在维护人员的子系统树中，并进入 -next 树（如下所述）。当流程进行时，此步骤将会对补丁进行更广泛的审查，并发现由于将此补丁与其他人所做的工作合并而导致的任何问题。
- 请注意，大多数维护人员也有日常工作，因此合并补丁可能不是他们的最优先工作。如果您的补丁得到了需要更改的反馈，那么您应该进行这些更改，或者解释为何不应该进行这些更改。如果您的补丁没有评审意见，也没有被其相应的子系统或驱动程序维护者接受，那么您应该坚持不懈地将补丁更新到当前内核使其可被正常应用，并不断地发送它以供审查和合并。
- 合并到主线。最终，一个成功的补丁将被合并到由 Linus Torvalds 管理的主线存储库中。此时可能会出现更多的评论和/或问题；对开发人员来说应对这些问题并解决出现的任何问题仍很重要。
- 稳定版发布。大量用户可能受此补丁影响，因此可能再次出现新的问题。
- 长期维护。虽然开发人员在合并代码后可能会忘记代码，但这种行为往往会给开发社区留下不良印象。合并代码消除了一些维护负担，因为其他人将修复由 API 更改引起的问题。但是，如果代码要长期保持

可用，原始开发人员应该继续为代码负责。

内核开发人员（或他们的雇主）犯的最大错误之一是试图将流程简化为一个“合并到主线”步骤。这种方法总是会让所有相关人员感到沮丧。

## 补丁如何进入内核

只有一个人可以将补丁合并到主线内核存储库中：Linus Torvalds。但是，在进入 2.6.38 内核的 9500 多个补丁中，只有 112 个（大约 1.3%）是由 Linus 自己直接选择的。内核项目已经发展到一个没有一个开发人员可以在没有支持的情况下检查和选择每个补丁的规模。内核开发人员处理这种增长的方式是使用围绕信任链构建的助理系统。

内核代码库在逻辑上被分解为一组子系统：网络、特定体系结构支持、内存管理、视频设备等。大多数子系统都有一个指定的维护人员，其总体负责该子系统中的代码。这些子系统维护者（松散地）是他们所管理的内核部分的“守门员”；他们（通常）会接受一个补丁以包含到主线内核中。

子系统维护人员每个人都管理着自己版本的内核源代码树，通常（并非总是）使用 Git。Git 等工具（以及 Quilt 或 Mercurial 等相关工具）允许维护人员跟踪补丁列表，包括作者信息和其他元数据。在任何给定的时间，维护人员都可以确定他或她的存储库中的哪些补丁在主线中找不到。

当合并窗口打开时，顶级维护人员将要求 Linus 从存储库中“拉出”他们为合并选择的补丁。如果 Linus 同意，补丁流将流向他的存储库，成为主线内核的一部分。Linus 对拉取中接收到的特定补丁的关注程度各不相同。很明显，有时他看起来很关注。但是一般来说，Linus 相信子系统维护人员不会向上游发送坏补丁。

子系统维护人员反过来也可以从其他维护人员那里获取补丁。例如，网络树是由首先在专用于网络设备驱动程序、无线网络等的树中积累的补丁构建的。此存储链可以任意长，但很少超过两个或三个链接。由于链中的每个维护者都信任那些管理较低级别树的维护者，所以这个过程称为“信任链”。

显然，在这样的系统中，获取内核补丁取决于找到正确的维护者。直接向 Linus 发送补丁通常不是正确的方法。

## Next 树

子系统树链引导补丁流到内核，但它也提出了一个有趣的问题：如果有人想查看为下一个合并窗口准备的所有补丁怎么办？开发人员将感兴趣的是，还有什么其他的更改有待解决，以了解是否存在需要担心的冲突；例如，更改核心内核函数原型的修补程序将与使用该函数旧形式的任何其他修补程序冲突。审查人员和测试人员希望在所有这些变更到达主线内核之前，能够访问它们的集成形式的变更。您可以从所有相关的子系统树中提取更改，但这将是一项复杂且容易出错的工作。

解决方案以-next 树的形式出现，在这里子系统树被收集以供测试和审查。这些树中由 Andrew Morton 维护的一个，被称为“-mm”（用于内存管理，创建时为此）。-mm 树集成了一长串子系统树中的补丁；它还包含一些旨在帮助调试的补丁。

除此之外，-mm 还包含大量由 Andrew 直接选择的补丁。这些补丁可能已经发布在邮件列表上，或者它们可能应用于内核中未指定子系统树的部分。同时，-mm 作为最后手段的子系统树；如果没有其他明显的路径可以让补丁进入主线，那么它很可能最终选择-mm 树。累积在-mm 中的各种补丁最终将被转发到适当的子系统树，或者直接发送到 Linus。在典型的开发周期中，大约 5-10% 的补丁通过-mm 进入主线。

当前-mm 补丁可在“mmotm”(-mm of the moment) 目录中找到:

<https://www.ozlabs.org/~akpm/mmotm/>

然而, 使用 MMOTM 树可能会十分令人头疼; 它甚至可能无法编译。

下一个周期补丁合并的主要树是 linux-next, 由 Stephen Rothwell 维护。根据设计 linux-next 是下一个合并窗口关闭后主线的快照。linux-next 树在 Linux-kernel 和 Linux-next 邮件列表中发布, 可从以下位置下载:

<https://www.kernel.org/pub/linux/kernel/next/>

Linux-next 已经成为内核开发过程中不可或缺的一部分; 在一个给定的合并窗口中合并的所有补丁都应该在合并窗口打开之前的一段时间内找到进入 Linux-next 的方法。

## Staging 树

内核源代码树包含 drivers/staging/ 目录, 其中有许多驱动程序或文件系统的子目录正在被添加到内核树中。它们在仍然需要更多的修正的时候可以保留在 driver/staging/ 目录中; 一旦完成, 就可以将它们移到内核中。这是一种跟踪不符合 Linux 内核编码或质量标准的驱动程序的方法, 人们可能希望使用它们并跟踪开发。

Greg Kroah Hartman 目前负责维护 staging 树。仍需要修正的驱动程序将发送给他, 每个驱动程序在 drivers/staging/ 中都有自己的子目录。除了驱动程序源文件之外, 目录中还应该有一个 TODO 文件。TODO 文件列出了驱动程序需要接受的暂停的工作, 以及驱动程序的任何补丁都应该抄送的人员列表。当前的规则要求, staging 的驱动程序必须至少正确编译。

Staging 是一种让新的驱动程序进入主线的相对容易的方法, 它们会幸运地引起其他开发人员的注意, 并迅速改进。然而, 进入 staging 并不是故事的结尾; staging 中没有看到常规进展的代码最终将被删除。经销商也倾向于相对不愿意使用 staging 驱动程序。因此, 在成为一个合适的主线驱动的路上, staging 仅是一个中转站。

## 工具

从上面的文本可以看出, 内核开发过程在很大程度上依赖于在不同方向上聚集补丁的能力。如果没有适当强大的工具, 整个系统将无法在任何地方正常工作。关于如何使用这些工具的教程远远超出了本文档的范围, 但还是用一点篇幅介绍一些关键点。

到目前为止, 内核社区使用的主要源代码管理系统是 git。Git 是在自由软件社区中开发的许多分布式版本控制系统之一。它非常适合内核开发, 因为它在处理大型存储库和大量补丁时性能非常好。它也以难以学习和使用而著称, 尽管随着时间的推移它变得更好了。对于内核开发人员来说, 对 Git 的某种熟悉几乎是一种要求; 即使他们不将它用于自己的工作, 他们也需要 Git 来跟上其他开发人员(以及主线)正在做的事情。

现在几乎所有的 Linux 发行版都打包了 Git。Git 主页位于:

<https://git-scm.com/>

此页面包含了文档和教程的链接。

在不使用 git 的内核开发人员中, 最流行的选择几乎肯定也是 Mercurial:

<http://www.seleric.com/mercurial/>

Mercurial 与 Git 共享许多特性，但它提供了一个界面，许多人觉得它更易于使用。

另一个值得了解的工具是 Quilt：

<https://savannah.nongnu.org/projects/quilt>

Quilt 是一个补丁管理系统，而不是源代码管理系统。它不会随着时间的推移跟踪历史；相反，它面向根据不断发展的代码库跟踪一组特定的更改。一些主要的子系统维护人员使用 Quilt 来管理打算向上游移动的补丁。对于某些树的管理（例如-mm），quilt 是最好的工具。

## 邮件列表

大量的 Linux 内核开发工作是通过邮件列表完成的。如果不加入至少一个某个列表，就很难成为社区中的一个“全功能”成员。但是，Linux 邮件列表对开发人员来说也是一个潜在的危险，他们可能会被一堆电子邮件淹没、违反 Linux 列表上使用的约定，或者两者兼而有之。

大多数内核邮件列表都在 vger.kernel.org 上运行；主列表位于：

<http://vger.kernel.org/vger-lists.html>

不过，也有一些列表托管在别处；其中一些列表位于 [redhat.com/mailman/listinfo](http://redhat.com/mailman/listinfo)。

当然，内核开发的核心邮件列表是 linux-kernel。这个列表是一个令人生畏的地方：每天的信息量可以达到 500 条，噪音很高，谈话技术性很强，且参与者并不总是表现出高度的礼貌。但是，没有其他地方可以让内核开发社区作为一个整体聚集在一起；不使用此列表的开发人员将错过重要信息。

以下一些提示可以帮助在 linux-kernel 生存：

- 将邮件转移到单独的文件夹，而不是主邮箱文件夹。我们必须能够持续地忽略洪流。
- 不要试图跟上每一次谈话——没人会这样。重要的是要筛选感兴趣的主题（但请注意长时间的对话可能会偏离原来的主题，尽管未改变电子邮件的主题）和参与的人。
- 不要回复挑事的人。如果有人试图激起愤怒，请忽略他们。
- 当回复 Linux 内核电子邮件（或其他列表上的电子邮件）时，请为所有相关人员保留 Cc: 抄送头。如果没有确实的理由（如明确的请求），则不应删除收件人。一定要确保你要回复的人在抄送列表中。这个惯例也使你不必在回复邮件时明确要求被抄送。
- 在提出问题之前，搜索列表存档（和整个网络）。有些开发人员可能会对那些显然没有完成家庭作业的人感到不耐烦。
- 避免顶部回复（把你的答案放在你要回复的引文上面的做法）。这会让你的回答更难理解，印象也很差。
- 在正确的邮件列表发问。linux-kernel 可能是通用的讨论场所，但它不是寻找所有子系统开发人员的最佳场所。

最后一点——找到正确的邮件列表——是开发人员常出错的地方。在 linux-kernel 上提出与网络相关的问题的人几乎肯定会收到一个礼貌的建议，转到 netdev 列表上提出，因为这是大多数网络开发人员经常出现的列

表。还有其他列表可用于 scsi、video4linux、ide、filesystem 等子系统。查找邮件列表的最佳位置是与内核源代码一起打包的 MAINTAINERS 文件。

### 开始内核开发

关于如何开始内核开发过程的问题很常见——个人和公司皆然。同样常见的是失误，这使得关系的开始比本应的更困难。

公司通常希望聘请知名的开发人员来启动开发团队。实际上，这是一种有效的技术。但它也往往是昂贵的，而且对增加有经验的内核开发人员的数量没有多大帮助。考虑到时间投入，可以让内部开发人员加快 Linux 内核的开发速度。利用这段时间可以让雇主拥有一批既了解内核又了解公司的开发人员，还可以帮助培训其他人。从中期来看，这通常是更有利可图的方法。

可以理解的是，单个开发人员往往对起步感到茫然。从一个大型项目开始可能会很吓人；人们往往想先用一些较小的东西来试试水。由此，一些开发人员开始创建修补拼写错误或轻微编码风格问题的补丁。不幸的是，这样的补丁会产生一定程度的噪音，这会分散整个开发社区的注意力，因此，它们越来越被人不看重。希望向社区介绍自己的新开发人员将无法通过这些方式获得他们期待的反响。

Andrew Morton 为有抱负的内核开发人员提供了如下建议

所有内核开发者的第一个项目肯定应该是“确保内核在您可以操作的所有机器上始终完美运行”。通常的方法是和其他人一起解决问题（这可能需要坚持！），但就是如此——这是内核开发的一部分。

(<http://lwn.net/articles/283982/>)

在没有明显问题需要解决的情况下，通常建议开发人员查看当前的回归和开放缺陷列表。从来都不缺少需要解决的问题；通过解决这些问题，开发人员将从该过程获得经验，同时与开发社区的其他成员建立相互尊重。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

---

**Original** Documentation/process/3.Early-stage.rst

**Translator** 时奎亮 Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

校译 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

## 早期规划

当考虑一个 Linux 内核开发项目时，很可能会直接跳进去开始编码。然而，与任何重要的项目一样，许多成功的基础最好是在第一行代码编写之前就打下。在早期计划和沟通中花费一些时间可以在之后节省更多的时间。

## 搞清问题

与任何工程项目一样，成功的内核改善从清晰描述要解决的问题开始。在某些情况下，这个步骤很容易：例如当某个特定硬件需要驱动程序时。不过，在其他情况下，很容易将实际问题与建议的解决方案混在一起，这可能会导致麻烦。

举个例子：几年前，Linux 音频的开发人员寻求一种方法来运行应用程序，而不会因系统延迟过大而导致退出或其他问题。他们得到的解决方案是一个连接到 Linux 安全模块（LSM）框架中的内核模块；这个模块可以配置为允许特定的应用程序访问实时调度程序。这个模块被实现并发到 `linux-kernel` 邮件列表，在那里它立即遇到了麻烦。

对于音频开发人员来说，这个安全模块足以解决他们当前的问题。但是，对于更广泛的内核社区来说，这被视为对 LSM 框架的滥用（LSM 框架并不打算授予他们原本不具备的进程特权），并对系统稳定性造成风险。他们首选的解决方案包括短期的通过 `rlimit` 机制进行实时调度访问，以及长期的减少延迟的工作。

然而，音频社区无法超越他们实施的特定解决方案来看问题；他们不愿意接受替代方案。由此产生的分歧使这些开发人员对整个内核开发过程感到失望；其中一个开发人员返回到 `audio` 列表并发布了以下内容：

有很多非常好的 Linux 内核开发人员，但他们往往会被一群傲慢的傻瓜所压倒。试图向这些人传达用户需求是浪费时间。他们太“聪明”了，根本听不到少数人的话。

(<http://lwn.net/articles/131776/>)

实际情况却是不同的；与特定模块相比，内核开发人员更关心系统稳定性、长期维护以及找到问题的正确解决方案。这个故事的寓意是把重点放在问题上——而不是具体的解决方案上——并在开始编写代码之前与开发社区讨论这个问题。

因此，在考虑一个内核开发项目时，我们应该得到一组简短问题的答案：

- 需要解决的问题究竟是什么？
- 受此问题影响的用户有哪些？解决方案应该解决哪些使用案例？
- 内核现在为何没能解决这个问题？

只有这样，才能开始考虑可能的解决方案。

### 早期讨论

在计划内核开发项目时，在开始实施之前与社区进行讨论是很有意义的。早期沟通可以通过多种方式节省时间和麻烦：

- 很可能问题是由内核以您不理解的方式解决的。Linux 内核很大，具有许多不明显的特性和功能。并不是所有的内核功能都像人们所希望的那样有文档记录，而且很容易遗漏一些东西。某作者发布了一个完整的驱动程序，重复了一个其不知道的现有驱动程序。重新发明现有轮子的代码不仅浪费，而且不会被接受到主线内核中。
- 建议的解决方案中可能有一些要素不适合并入主线。在编写代码之前，最好先了解这样的问题。
- 其他开发人员完全有可能考虑过这个问题；他们可能有更好的解决方案的想法，并且可能愿意帮助创建这个解决方案。

在内核开发社区的多年经验给了我们一个明确的教训：闭门设计和开发的内核代码总是有一些问题，这些问题只有在代码发布到社区中时才会被发现。有时这些问题很严重，需要数月或数年的努力才能使代码达到内核社区的标准。例如：

- 设计并实现了单处理器系统的 DeviceScape 网络栈。只有使其适合于多处理器系统，才能将其合并到主线中。在代码中修改锁等等是一项困难的任务；因此，这段代码（现在称为 mac80211）的合并被推迟了一年多。
- Reiser4 文件系统包含许多功能，核心内核开发人员认为这些功能应该在虚拟文件系统层中实现。它还包括一些特性，这些特性在不将系统暴露于用户引起的死锁的情况下是不容易实现的。这些问题过迟发现——以及拒绝处理其中一些问题——已经导致 Reiser4 置身主线内核之外。
- Apparmor 安全模块以被认为不安全和不可靠的方式使用内部虚拟文件系统数据结构。这种担心（包括其他）使 Apparmor 多年来无法进入主线。

在这些情况下，与内核开发人员的早期讨论，可以避免大量的痛苦和额外的工作。

### 找谁交流？

当开发人员决定公开他们的计划时，下一个问题是：我们从哪里开始？答案是找到正确的邮件列表和正确的维护者。对于邮件列表，最好的方法是在维护者 (MAINTAINERS) 文件中查找要发布的相关位置。如果有一个合适的子系统列表，那么其上发布通常比在 linux-kernel 上发布更可取；您更有可能接触到在相关子系统中具有专业知识的开发人员，并且环境可能具支持性。

找到维护人员可能会有点困难。同样，维护者文件是开始的地方。但是，该文件往往不是最新的，并且并非所有子系统都在那里显示。实际上，维护者文件中列出的人员可能不是当前实际担任该角色的人员。因此，当对联系谁有疑问时，一个有用的技巧是使用 git（尤其是“git-log”）查看感兴趣的子系统中当前活动的用户。看看谁在写补丁、谁会在这些补丁上加上 Signed-off-by 行签名（如有）。这些人将是帮助新开发项目的最佳人选。

找到合适的维护者有时是非常具有挑战性的，以至于内核开发人员添加了一个脚本来简化这个过程：

```
.../scripts/get_maintainer.pl
```

当给定“-f”选项时，此脚本将返回指定文件或目录的当前维护者。如果在命令行上给出了一个补丁，它将列出可能接收补丁副本的维护人员。有许多选项可以调节 `get_maintainer.pl` 搜索维护者的严格程度；请小心使用更激进的选项，因为最终结果可能会包括对您正在修改的代码没有真正兴趣的开发人员。

如果所有其他方法都失败了，那么与 Andrew Morton 交流是跟踪特定代码段维护人员的一种有效方法。

## 何时邮寄？

如果可能的话，在早期阶段发布你的计划只会更有帮助。描述正在解决的问题以及已经制定的关于如何实施的任何计划。您可以提供的任何信息都可以帮助开发社区为项目提供有用的输入。

在这个阶段可能发生的一件令人沮丧的事情不是得到反对意见，而是很少或根本没有反馈。令人伤心的事实是：(1) 内核开发人员往往很忙；(2) 不缺少有宏伟计划但代码（甚至代码设想）很少的人去支持他们；(3) 没有人有义务审查或评论别人发表的想法。除此之外，高层级的设计常常隐藏着一些问题，这些问题只有在有人真正尝试实现这些设计时才会被发现；因此，内核开发人员宁愿看到代码。

如果发布请求评论（RFC）并没得到什么有用的评论，不要以为这意味着无人对此项目有兴趣，同时你也不能假设你的想法没有问题。在这种情况下，最好的做法是继续进行，把你的进展随时通知社区。

## 获得官方认可

如果您的工作是在公司环境中完成的，就像大多数 Linux 内核工作一样；显然，在您将公司的计划或代码发布到公共邮件列表之前，必须获得有适当权利经理的许可。发布不确定是否兼容 GPL 的代码尤其会带来问题；公司的管理层和法律人员越早能够就发布内核开发项目达成一致，对参与的每个人都越好。

一些读者可能会认为他们的核心工作是为了支持还没有正式承认存在的产品。将雇主的计划公布在公共邮件列表上可能不是一个可行的选择。在这种情况下，有必要考虑保密是否真的是必要的；通常不需要把开发计划关在门内。

的确，有些情况下一家公司在开发过程的早期无法合法地披露其计划。拥有经验丰富的内核开发人员的公司可能选择以开环的方式进行开发，前提是他们以后能够避免严重的集成问题。对于没有这种内部专业知识的公司，最好的选择往往是聘请外部开发者根据保密协议审查计划。Linux 基金会运行了一个 NDA 程序，旨在帮助解决这种情况；更多信息参见：

<http://www.linuxfoundation.org/nda/>

这种审查通常足以避免以后出现严重问题，而无需公开披露项目。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:<alexs@kernel.org>)。

**Original** Documentation/process/4.Coding.rst

**Translator** 时奎亮 Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

校译 吴想成 Wu XiangCheng <[bowxc@email.cn](mailto:bowxc@email.cn)>

### 使代码正确

虽然一个坚实的、面向社区的设计过程有很多值得说道的，但是任何内核开发项目工作的证明都反映在代码中。它是将由其他开发人员检查并合并（或不合并）到主线树中的代码。所以这段代码的质量决定了项目的最终成功。

本节将检查编码过程。我们将从内核开发人员常犯的几种错误开始。然后重点将转移到正确的做法和相关有用的工具上。

### 陷阱

#### 代码风格

内核长期以来都有其标准的代码风格，如[Documentation/translations/zh\\_CN/process/coding-style.rst](Documentation/translations/zh_CN/process/coding-style.rst)中所述。在多数时候，该文档中描述的准则至多被认为是建议性的。因此，内核中存在大量不符合代码风格准则的代码。这种代码的存在会给内核开发人员带来两方面的危害。

首先，相信内核代码标准并不重要，也不强制执行。但事实上，如果没有按照标准编写代码，那么新代码将很难加入到内核中；许多开发人员甚至会在审查代码之前要求对代码进行重新格式化。一个像内核这么大的代码库需要一些统一格式的代码，以使开发人员能够快速理解其中的任何部分。所以再也经不起奇怪格式的代码的折腾了。

内核的代码风格偶尔会与雇主的强制风格发生冲突。在这种情况下，必须在代码合并之前遵从内核代码风格。将代码放入内核意味着以多种方式放弃一定程度的控制权——包括控制代码样式。

另一个危害是认为已经在内核中的代码迫切需要修复代码样式。开发者可能会开始编写重新格式化补丁，作为熟悉开发过程的一种方式，或者作为将其名字写入内核变更日志的一种方式，或者两者兼而有之。但是纯代码风格的修复被开发社区视为噪音，它们往往受到冷遇。因此，最好避免编写这种类型的补丁。在由于其他原因处理一段代码的同时顺带修复其样式是很自然的，但是不应该仅为了更改代码样式而更改之。

代码风格文档也不应该被视为绝对不可违反的规则。如果有一个足够的理由反对这种样式（例如为了 80 列限制拆分行会导致可读性大大降低），那么就这样做吧。

注意您还可以使用 `clang-format` 工具来帮助您处理这些规则，快速自动重新格式化部分代码，和审阅完整的文件以发现代码样式错误、拼写错误和可能的改进。它还可以方便地排序 `#includes`、对齐变量/宏、重排文本和其他类似任务。有关详细信息，请参阅文档 <Documentation/process/clang-format.rst>

## 抽象层

计算机科学教授教学生以灵活性和信息隐藏的名义广泛使用抽象层。当然，内核广泛地使用了抽象；任何涉及数百万行代码的项目都必须做到这一点以存续下来。但经验表明，过度或过早的抽象可能和过早的优化一样有害。抽象应用在适当层级，不要过度。

简单点，先考虑一个调用时始终只有一个参数且总为零的函数。我们可以保留这个参数，以在需要使用它时提供的额外灵活性。不过，在那时实现了这个额外参数的代码很有可能以某种从未被注意到的微妙方式被破坏——因为它从未被使用过。或者当需要额外的灵活性时，它并未以符合程序员当初期望的方式来实现。内核开发人员通常会提交补丁来删除未使用的参数；一般来说，一开始就不应该添加这些参数。

隐藏硬件访问的抽象层——通常为了允许大量的驱动程序兼容多个操作系统——尤其不受欢迎。这样的层使代码变得模糊，可能会造成性能损失；它们不属于 Linux 内核。

另一方面，如果您发现自己从另一个内核子系统复制了大量的代码，那么是时候了解一下：是否需要将这些代码中的部分提取到单独的库中，或者在更高的层次上实现这些功能。在整个内核中复制相同的代码没有价值。

## #ifdef 和预处理

C 预处理器似乎给一些 C 程序员带来了强大的诱惑，他们认为它是一种将大量灵活性加入源代码中的方法。但是预处理器不是 C，大量使用它会导致代码对其他人来说更难阅读，对编译器来说更难检查正确性。使用了大量预处理器几乎总是代码需要一些清理工作的标志。

使用 #ifdef 的条件编译实际上是一个强大的功能，它在内核中使用。但是很少有人希望看到代码被铺满 #ifdef 块。一般规定，ifdef 的使用应尽可能限制在头文件中。条件编译代码可以限制函数，如果代码不存在，这些函数就直接变成空的。然后编译器将悄悄地优化对空函数的调用。使得代码更加清晰，更容易理解。

C 预处理器宏存在许多危险性，包括可能对具有副作用且没有类型安全的表达式进行多重评估。如果您试图定义宏，请考虑创建一个内联函数替代。结果相同的代码，内联函数更容易阅读，不会多次计算其参数，并且允许编译器对参数和返回值执行类型检查。

## 内联函数

不过，内联函数本身也存在风险。程序员可以倾心于避免函数调用和用内联函数填充源文件所固有的效率。然而，这些功能实际上会降低性能。因为它们的代码在每个调用站点都被复制一遍，所以最终会增加编译内核的大小。此外，这也对处理器的内存缓存造成压力，从而大大降低执行速度。通常内联函数应该非常小，而且相对较少。毕竟函数调用的成本并不高；大量创建内联函数是过早优化的典型例子。

一般来说，内核程序员会自冒风险忽略缓存效果。在数据结构课程开头中的经典时间/空间权衡通常不适用于当代硬件。空间 就是时间，因为一个大的程序比一个更紧凑的程序运行得慢。

较新的编译器越来越激进地决定一个给定函数是否应该内联。因此，随意放置使用“inline”关键字可能不仅仅是过度的，也可能是无用的。

### 锁

2006 年 5 月，“deviceescape” 网络堆栈在前呼后拥下以 GPL 发布，并被纳入主线内核。这是一个受欢迎的消息；Linux 中对无线网络的支持充其量被认为是不合格的，而 Deviceescape 堆栈承诺修复这种情况。然而直到 2007 年 6 月（2.6.22），这段代码才真正进入主线。发生了什么？

这段代码出现了许多闭门造车的迹象。但一个大麻烦是，它并不是为多处理器系统而设计。在合并这个网络堆栈（现在称为 mac80211）之前，需要对其进行一个锁方案的改造。

曾经，Linux 内核代码可以在不考虑多处理器系统所带来的并发性问题的情况下进行开发。然而现在，这个文档就是在双核笔记本电脑上写的。即使在单处理器系统上，为提高响应能力所做的工作也会提高内核内的并发性水平。编写内核代码而不考虑锁的日子早已远去。

可以由多个线程并发访问的任何资源（数据结构、硬件寄存器等）必须由锁保护。新的代码应该谨记这一要求；事后修改锁是一项相当困难的任务。内核开发人员应该花时间充分了解可用的锁原语，以便为工作选择正确的工具。对并发性缺乏关注的代码很难进入主线。

### 回归

最后一个值得一提的危险是回归：它可能会引起导致现有用户的某些东西中断的改变（这也可能会带来很大的改进）。这种变化被称为“回归”，回归已经成为主线内核最不受欢迎的问题。除了少数例外情况，如果回归不能及时修正，会导致回归的修改将被取消。最好首先避免回归发生。

人们常常争论，如果回归带来的功能远超过产生的问题，那么回归是否为可接受的。如果它破坏了一个系统却为十个系统带来新的功能，为何不改改态度呢？2007 年 7 月，Linus 对这个问题给出了最佳答案：

所以我们不会通过引入新问题来修复错误。这种方式是靠不住的，没人知道是否真的有进展。是前进两步、后退一步，还是前进一步、后退两步？

[\(http://lwn.net/articles/243460/\)](http://lwn.net/articles/243460/)

特别不受欢迎的一种回归类型是用户空间 ABI 的任何变化。一旦接口被导出到用户空间，就必须无限期地支持它。这一事实使得用户空间接口的创建特别具有挑战性：因为它们不能以不兼容的方式进行更改，所以必须一次就对。因此，用户空间接口总是需要大量的思考、清晰的文档和广泛的审查。

### 代码检查工具

至少目前，编写无错误代码仍然是我们中很少人能达到的理想状态。不过，我们希望做的是，在代码进入主线内核之前，尽可能多地捕获并修复这些错误。为此，内核开发人员已经提供了一系列令人印象深刻的工具，可以自动捕获各种各样的隐藏问题。计算机发现的任何问题都是一个以后不会困扰用户的问题，因此，只要有可能，就应该使用自动化工具。

第一步是注意编译器产生的警告。当前版本的 GCC 可以检测（并警告）大量潜在错误。通常，这些警告都指向真正的问题。提交以供审阅的代码一般不会产生任何编译器警告。在消除警告时，注意了解真正的原因，并尽量避免仅“修复”使警告消失而不解决其原因。

请注意，并非所有编译器警告都默认启用。使用“make KCFLAGS=-W”构建内核以获得完整集合。

内核提供了几个配置选项，可以打开调试功能；大多数配置选项位于“kernel hacking”子菜单中。对于任何用于开发或测试目的的内核，都应该启用其中几个选项。特别是，您应该打开：

- FRAME\_WARN 获取大于给定数量的堆栈帧的警告。这些警告生成的输出可能比较冗长，但您不必担心来自内核其他部分的警告。
- DEBUG\_OBJECTS 将添加代码以跟踪内核创建的各种对象的生命周期，并在出现问题时发出警告。如果你要添加创建（和导出）关于其自己的复杂对象的子系统，请考虑打开对象调试基础结构的支持。
- DEBUG\_SLAB 可以发现各种内存分配和使用错误；它应该用于大多数开发内核。
- DEBUG\_SPINLOCK, DEBUG\_ATOMIC\_SLEEP 和 DEBUG\_MUTEXES 会发现许多常见的锁错误。

还有很多其他调试选项，其中一些将在下面讨论。其中一些有显著的性能影响，不应一直使用。在学习可用选项上花费一些时间，可能会在短期内得到许多回报。

其中一个较重的调试工具是锁检查器或“lockdep”。该工具将跟踪系统中每个锁（spinlock 或 mutex）的获取和释放、获取锁的相对顺序、当前中断环境等等。然后，它可以确保总是以相同的顺序获取锁，相同的中断假设适用于所有情况等等。换句话说，lockdep 可以找到许多导致系统死锁的场景。在部署的系统中，这种问题可能会很痛苦（对于开发人员和用户而言）；LockDep 允许提前以自动方式发现问题。具有任何类型的非普通锁的代码在提交合并前应在启用 lockdep 的情况下运行测试。

作为一个勤奋的内核程序员，毫无疑问，您将检查任何可能失败的操作（如内存分配）的返回状态。然而，事实上，最终的故障复现路径可能完全没有经过测试。未测试的代码往往会出现问题；如果所有这些错误处理路径都被执行了几次，那么您可能对代码更有信心。

内核提供了一个可以做到这一点的错误注入框架，特别是在涉及内存分配的情况下。启用故障注入后，内存分配的可配置失败的百分比；这些失败可以限定在特定的代码范围内。在启用了故障注入的情况下运行，程序员可以看到当情况恶化时代码如何响应。有关如何使用此工具的详细信息，请参阅 Documentation/fault-injection/fault-injection.rst。

“sparse”静态分析工具可以发现其他类型的错误。sparse 可以警告程序员用户空间和内核空间地址之间的混淆、大端序与小端序的混淆、在需要一组位标志的地方传递整数值等等。sparse 必须单独安装（如果您的分发服务器没有将其打包，可以在 [https://sparse.wiki.kernel.org/index.php/Main\\_page](https://sparse.wiki.kernel.org/index.php/Main_page) 找到），然后可以通过在 make 命令中添加“C=1”在代码上运行它。

“Coccinelle”工具 <http://coccinelle.lip6.fr/> 能够发现各种潜在的编码问题；它还可以为这些问题提出修复方案。在 scripts/coccinelle 目录下已经打包了相当多的内核“语义补丁”；运行“make coccicheck”将运行这些语义补丁并报告发现的任何问题。有关详细信息，请参阅 Documentation/dev-tools/coccinelle.rst

其他类型的可移植性错误最好通过为其他体系结构编译代码来发现。如果没有 S/390 系统或 Blackfin 开发板，您仍然可以执行编译步骤。可以在以下位置找到一大堆用于 x86 系统的交叉编译器：

<https://www.kernel.org/pub/tools/crosstool/>

花一些时间安装和使用这些编译器将有助于避免以后的尴尬。

### 文档

文档通常比内核开发规则更为例外。即便如此，足够的文档将有助于简化将新代码合并到内核中的过程，使其他开发人员的生活更轻松，并对您的用户有所帮助。在许多情况下，添加文档已基本上是强制性的。

任何补丁的第一个文档是其关联的变更日志。日志条目应该描述正在解决的问题、解决方案的形式、处理补丁的人员、对性能的任何相关影响，以及理解补丁可能需要的任何其他内容。确保变更日志说明了 \* 为什么 \* 补丁值得应用；大量开发者未能提供这些信息。

任何添加新用户空间接口的代码——包括新的 sysfs 或/proc 文件——都应该包含该接口的文档，该文档使用户空间开发人员能够知道他们在使用什么。请参阅 Documentation/ABI/README，了解如何此文档格式以及需要提供哪些信息。

文档 Documentation/admin-guide/kernel-parameters.rst 描述了内核的所有引导时间参数。任何添加新参数的补丁都应该向该文档添加适当的条目。

任何新的配置选项都必须附有帮助文本，帮助文本需清楚地解释这些选项以及用户可能希望何时使用它们。

许多子系统的内部 API 信息通过专门格式化的注释进行记录；这些注释可以通过“kernel-doc”脚本以多种方式提取和格式化。如果您在具有 kerneldoc 注释的子系统中工作，则应该维护它们，并根据需要为外部可用的功能添加它们。即使在没有如此记录的领域中，为将来添加 kerneldoc 注释也没有坏处；实际上，这对于刚开始开发内核的人来说是一个有用的活动。这些注释的格式以及如何创建 kerneldoc 模板的一些信息可以在 Documentation/doc-guide/ 上找到。

任何阅读大量现有内核代码的人都会注意到，注释的缺失往往是最值得注意的。同时，对新代码的要求比过去更高；合并未注释的代码将更加困难。这就是说，人们并不期望详细注释的代码。代码本身应该是自解释的，注释阐释了更微妙的方面。

某些事情应该总是被注释。使用内存屏障时，应附上一行文字，解释为什么需要设置内存屏障。数据结构的锁规则通常需要在某个地方解释。一般来说，主要数据结构需要全面的文档。应该指出代码中分立的位之间不明显的依赖性。任何可能诱使代码管理人进行错误的“清理”的事情都需要一个注释来说明为什么要这样做。等等。

### 内部 API 更改

内核提供给用户空间的二进制接口不能被破坏，除非逼不得已。而内核的内部编程接口是高度流动的，当需要时可以更改。如果你发现自己不得不处理一个内核 API，或者仅仅因为它不满足你的需求导致无法使用特定的功能，这可能是 API 需要改变的一个标志。作为内核开发人员，您有权进行此类更改。

的确可以进行 API 更改，但更改必须是合理的。因此任何进行内部 API 更改的补丁都应该附带关于更改内容和必要原因的描述。这种变化也应该拆分成一个单独的补丁，而不是埋在一个更大的补丁中。

另一个要点是，更改内部 API 的开发人员通常要负责修复内核树中被更改破坏的任何代码。对于一个广泛使用的函数，这个责任可以导致成百上千的变化，其中许多变化可能与其他开发人员正在做的工作相冲突。不用说，这可能是一项大工程，所以最好确保理由是可靠的。请注意，coccinelle 工具可以帮助进行广泛的 API 更改。

在进行不兼容的 API 更改时，应尽可能确保编译器捕获未更新的代码。这将帮助您确保找到该接口的树内用处。它还将警告开发人员树外代码存在他们需要响应的更改。支持树外代码不是内核开发人员需要担心的事情，

但是我们也不必使树外开发人员的生活有不必要的困难。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:<alexs@kernel.org>)。

---

**Original** Documentation/process/5.Posting.rst

**Translator** 时奎亮 Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

校译 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

## 发布补丁

您的 工作迟早会准备 好提交给 社区进行 审查，并 最终包含 到 主线 内核 中。毫不 稀奇，内核 开发 社区已经 发展出 一套 用于发布 补丁 的 约定 和 过程；遵循 这些 约定 和 过程 将使 参与 其中的 每个人 的 生活 更加 轻松。本 文档 尝试 描述 这些 约定 的 部分 细节；更多 信息 也 可在 以下 文档 中 找到 *Documentation/translations/zh\_CN/process/submitting-patches.rst*, *Documentation/translations/zh\_CN/process/submitting-drivers.rst* 和 *Documentation/translations/zh\_CN/process/submit-checklist.rst*。

## 何时寄送

在 补丁 完全 “准备 好” 之前，避免 发布 补丁 是一种 持续的 诱惑。对于 简单的 补丁，这 不是 问题。但 是如果 正在 完成 的 工作 很 复杂，那么 在 工作 完成 之前 从 社区 获得 反馈 就 可以 获得 很多 好处。因此，您 应该 考虑 发布 正在 进行 的 工作，甚 至 维护 一个 可用 的 Git 树，以便 感兴趣 的 开发 人员 可以 随时 赶上 您的工作。

当 发布 中有 尚未 准备 好被 包含 的 代码，最好 在 发布 中 说明。还 应 提及 任何 有待 完成 的 主要 工作 和 任何 已知 问题。很少 有人 会 愿意 看 那些 被 认为 是 半生 不熟 的 补丁，但是 那些 愿意 的 人 会 带着 他们的 点子 来 一起 帮助 你 把 工作 推向 正确 的 方向。

### 创建补丁之前

在考虑将补丁发送到开发社区之前，有许多事情应该做。包括：

- 尽可能地测试代码。利用内核的调试工具，确保内核使用了所有可能的配置选项组合进行构建，使用交叉编译器为不同的体系结构进行构建等。
- 确保您的代码符合内核代码风格指南。
- 您的更改是否具有性能影响？如果是这样，您应该运行基准测试来显示您的变更的影响（或好处）；结果的摘要应该包含在补丁中。
- 确保您有权发布代码。如果这项工作是为雇主完成的，雇主对这项工作具有所有权，并且必须同意根据 GPL 对其进行发布。

一般来说，在发布代码之前进行一些额外的思考，几乎总是能在短时间内得到回报。

### 补丁准备

准备补丁发布的工作量可能很惊人，但在此尝试节省时间通常是不明智的，即使在短期内亦然。

必须针对内核的特定版本准备补丁。一般来说，补丁应该基于 Linus 的 Git 树中的当前主线。当以主线为基础时，请从一个众所周知的发布点开始——如稳定版本或 -rc 版本发布点——而不是在一个任意的主线分支点。

也可能需要针对-mm、linux-next 或子系统树生成版本，以便于更广泛的测试和审查。根据补丁的区域以及其他地方的情况，针对其他树建立的补丁可能需要大量的工作来解决冲突和处理 API 更改。

只有最简单的更改才应格式化为单个补丁；其他所有更改都应作为一系列逻辑更改进行。分割补丁是一门艺术；一些开发人员花了很长时间来弄清楚如何按照社区期望的方式来分割。不过，这些经验法则也许有帮助：

- 您发布的补丁系列几乎肯定不会是开发过程中版本控制系统中的一系列更改。相反，需要对您所做更改的最终形式加以考虑，然后以有意义的方式进行拆分。开发人员对离散的、自包含的更改感兴趣，而不是您创造这些更改的原始路径。
- 每个逻辑上独立的变更都应该格式化为单独的补丁。这些更改可以是小的（如“向此结构体添加字段”）或大的（如添加一个重要的新驱动程序），但它们在概念上应该是小的，并且可以在一行内简述。每个补丁都应做一个特定的、可以单独检查并验证它所做的事情的更改。
- 换种方式重申上述准则，也就是说：不要在同一补丁中混合不同类型的更改。如果一个补丁修复了一个关键的安全漏洞，又重新排列了一些结构，还重新格式化了代码，那么它很有可能会被忽略，从而导致重要的修复丢失。
- 每个补丁都应能创建一个可以正确地构建和运行的内核；如果补丁系列在中间被断开，那么结果仍应是一个正常工作的内核。部分应用一系列补丁是使用“git bisect”工具查找回归的一个常见场景；如果结果是一个损坏的内核，那么将使那些从事追踪问题的高尚工作的开发人员和用户的生活更加艰难。
- 不要过分分割。一位开发人员曾经将一组针对单个文件的编辑分成 500 个单独的补丁发布，这并没有使他成为内核邮件列表中最受欢迎的人。一个补丁可以相当大，只要它仍然包含一个单一的逻辑变更。

- 用一系列补丁添加一个全新的基础设施，但是该设施在系列中的最后一个补丁启用整个变更之前不能使用，这看起来很诱人。如果可能的话，应该避免这种诱惑；如果这个系列增加了回归，那么二分法将指出最后一个补丁是导致问题的补丁，即使真正的 bug 在其他地方。只要有可能，添加新代码的补丁程序应该立即激活该代码。

创建完美补丁系列的工作可能是一个令人沮丧的过程，在完成“真正的工作”之后需要花费大量的时间和思考。但是如果做得好，花费的时间就是值得的。

## 补丁格式和更改日志

所以现在你有了一系列完美的补丁可以发布，但是这项工作还没有完成。每个补丁都需要被格式化成一条消息，以快速而清晰地将其目的传达到世界其他地方。为此，每个补丁将由以下部分组成：

- 可选的“From”行，表明补丁作者。只有当你通过电子邮件发送别人的补丁时，这一行才是必须的，但是为防止疑问加上它也不会有什么坏处。
- 一行描述，说明补丁的作用。对于在没有其他上下文的情况下看到该消息的读者来说，该消息应足以确定修补程序的范围；此行将显示在“short form（简短格式）”变更日志中。此消息通常需要先加上子系统名称前缀，然后是补丁的目的。例如：

```
gpio: fix build on CONFIG_GPIO_SYSFS=n
```

- 一行空白，后接补丁内容的详细描述。此描述可以是任意需要的长度；它应该说明补丁的作用以及为什么它应该应用于内核。
- 一个或多个标记行，至少有一个由补丁作者的 Signed-off-by 签名。标记将在下面详细描述。

上面的项目一起构成补丁的变更日志。写一则好的变更日志是一门至关重要但常常被忽视的艺术；值得花一点时间来讨论这个问题。当你编写变更日志时，你应该记住有很多不同的人会读你的话。其中包括子系统维护人员和审查人员，他们需要决定是否应该合并补丁，分销商和其他维护人员试图决定是否应该将补丁反向移植到其他内核，缺陷搜寻人员想知道补丁是否导致他们正在追查的问题，以及想知道内核如何变化的用户等等。一个好的变更日志以最直接和最简洁的方式向所有这些人传达所需的信息。

在结尾，总结行应该描述变更的影响和动机，以及在一行约束条件下可能发生的变化。然后，详细的描述可以详述这些主题，并提供任何需要的附加信息。如果补丁修复了一个缺陷，请引用引入该缺陷的提交（如果可能，请在引用提交时同时提供其 id 和标题）。如果某个问题与特定的日志或编译器输出相关联，请包含该输出以帮助其他人搜索同一问题的解决方案。如果更改是为了支持以后补丁中的其他更改，那么应当说明。如果更改了内部 API，请详细说明这些更改以及其他开发人员应该如何响应。一般来说，你越把自己放在每个阅读你变更日志的人的位置上，变更日志（和内核作为一个整体）就越好。

不需要说，变更日志是将变更提交到版本控制系统时使用的文本。接下来将是：

- 补丁本身，采用统一的（“-u”）补丁格式。使用“-p”选项来 diff 将使函数名与更改相关联，从而使结果补丁更容易被其他人读取。

您应该避免在补丁中包括与更改不相关文件（例如，构建过程生成的文件或编辑器备份文件）。文档目录中的“dontdiff”文件在这方面有帮助；使用“-X”选项将其传递给 diff。

上面提到的标签（tag）用于描述各种开发人员如何与这个补丁的开发相关联。[Documentation/translations/zh\\_CN/process/submitting-patches.rst](#) 文档中对它们进行了详细描述；下面是一个简短的总结。每一行的格式如下：

```
tag: Full Name <email address> optional-other-stuff
```

常用的标签有：

- **Signed-off-by:** 这是一个开发人员的证明，证明他或她有权提交补丁以包含到内核中。这表明同意开发者来源认证协议，其全文见[Documentation/translations/zh\\_CN/process/submitting-patches.rst](#) 如果没有合适的签字，则不能合并到主线中。
- **Co-developed-by:** 声明补丁是由多个开发人员共同创建的；当几个人在一个补丁上工作时，它用于给出共同作者（除了 From: 所给出的作者之外）。由于 Co-developed-by: 表示作者身份，所以每个共同开发人，必须紧跟在相关合作作者的 Signed-off-by 之后。具体内容和示例见以下文件[Documentation/translations/zh\\_CN/process/submitting-patches.rst](#)
- **Acked-by:** 表示另一个开发人员（通常是相关代码的维护人员）同意补丁适合包含在内核中。
- **Tested-by:** 声明某人已经测试了补丁并确认它可以工作。
- **Reviewed-by:** 表示某开发人员已经审查了补丁的正确性；有关详细信息，请参阅[Documentation/translations/zh\\_CN/process/submitting-patches.rst](#)
- **Reported-by:** 指定报告此补丁修复的问题的用户；此标记用于表示感谢。
- **Cc:** 指定某人收到了补丁的副本，并有机会对此发表评论。

在补丁中添加标签时要小心：只有 Cc: 才适合在没有指定人员明确许可的情况下添加。

## 寄送补丁

在寄送补丁之前，您还需要注意以下几点：

- 您确定您的邮件发送程序不会损坏补丁吗？被邮件客户端更改空白或修饰了行的补丁无法被另一端接受，并且通常不会进行任何详细检查。如果有任何疑问，先把补丁寄给你自己，让你自己确定它是完好无损的。

[Documentation/translations/zh\\_CN/process/email-clients.rst](#) 提供了一些有用的提示，可以让特定的邮件客户端正常发送补丁。

- 你确定你的补丁没有荒唐的错误吗？您应该始终通过 scripts/checkpatch.pl 检查补丁程序，并解决它提出的问题。请记住，checkpatch.pl，虽然体现了对内核补丁应该是什么样的大量思考，但它并不比您聪明。如果修复 checkpatch.pl 给的问题会使代码变得更糟，请不要这样做。

补丁应始终以纯文本形式发送。请不要将它们作为附件发送；这使得审阅者在答复中更难引用补丁的部分。相反，只需将补丁直接放到您的消息中。

寄出补丁时，重要的是将副本发送给任何可能感兴趣的人。与其他一些项目不同，内核鼓励人们甚至错误地发送过多的副本；不要假定相关人员会看到您在邮件列表中的发布。尤其是，副本应发送至：

- 受影响子系统的维护人员。如前所述，维护人员文件是查找这些人员的首选地方。
- 其他在同一领域工作的开发人员，尤其是那些现在可能在那里工作的开发人员。使用 git 查看还有谁修改了您正在处理的文件，这很有帮助。
- 如果您对某错误报告或功能请求做出响应，也可以抄送原始发送人。
- 将副本发送到相关邮件列表，或者若无相关列表，则发送到 linux-kernel 列表。
- 如果您正在修复一个缺陷，请考虑该修复是否应进入下一个稳定更新。如果是这样，补丁副本也应发到 stable@vger.kernel.org。另外，在补丁本身的标签中添加一个“Cc: stable@vger.kernel.org”；这将使稳定版团队在修复进入主线时收到通知。

当为一个补丁选择接收者时，最好清楚你认为谁最终会接受这个补丁并将其合并。虽然可以将补丁直接发给 Linus Torvalds 并让他合并，但通常情况下不会这样做。Linus 很忙，并且有子系统维护人员负责监视内核的特定部分。通常您会希望维护人员合并您的补丁。如果没有明显的维护人员，Andrew Morton 通常是最后的补丁接收者。

补丁需要好的主题行。补丁主题行的规范格式如下：

[PATCH nn/mm] subsys: one-line description of the patch
---

其中“nn”是补丁的序号，“mm”是系列中补丁的总数，“subsys”是受影响子系统的名称。当然，一个单独的补丁可以省略 nn/mm。

如果您有一系列重要的补丁，那么通常发送一个简介作为第〇部分。不过，这个约定并没有得到普遍遵循；如果您使用它，请记住简介中的信息不会进入内核变更日志。因此，请确保补丁本身具有完整的变更日志信息。

一般来说，多部分补丁的第二部分和后续部分应作为对第一部分的回复发送，以便它们在接收端都连接在一起。像 git 和 coilt 这样的工具有命令，可以通过适当的线程发送一组补丁。但是，如果您有一长串补丁，并正使用 git，请不要使用--chain-reply-to 选项，以避免创建过深的嵌套。

<b>Warning:</b> 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。
--

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/process/6.Followthrough.rst

**Translator** 时奎亮 Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

校译 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

### 跟进

此时，您已经遵循了到目前为止给出的指导方针，并且，随着您自己的工程技能的增加，已经发布了一系列完美的补丁。即使是经验丰富的内核开发人员也能犯的最大错误之一是，认为他们的工作现在已经完成了。事实上，发布补丁意味着进入流程的下一个阶段，可能还需要做很多工作。

一个补丁在首次发布时就非常出色、没有改进的余地，这是很罕见的。内核开发流程已认识到这一事实，因此它非常注重对已发布代码的改进。作为代码的作者，您应该与内核社区合作，以确保您的代码符合内核的质量标准。如果不参与这个过程，很可能会无法将补丁合并到主线中。

### 与审阅者合作

任何意义上的补丁都会导致其他开发人员在审查代码时发表大量评论。对于许多开发人员来说，与审阅人员合作可能是内核开发过程中最令人生畏的部分。但是如果你记住一些事情，生活会变得容易得多：

- 如果你已经很好地解释了你的补丁，审阅人员会理解它的价值，以及为什么你会费尽心思去写它。但是这个并不能阻止他们提出一个基本的问题：在五年或十年后维护含有此代码的内核会怎么样？你可能被要求做出的许多改变——从编码风格的调整到大量的重写——都来自于对 Linux 的理解，即从现在起十年后，Linux 仍将在开发中。
- 代码审查是一项艰苦的工作，这是一项相对吃力不讨好的工作；人们记得谁编写了内核代码，但对于那些审查它的人来说，几乎没有什幺长久的名声。因此，审阅人员可能会变得暴躁，尤其是当他们看到同样的错误被一遍又一遍地犯下时。如果你得到了一个看起来愤怒、侮辱或完全冒犯你的评论，请抑制以同样方式回应的冲动。代码审查是关于代码的，而不是关于人的，代码审阅人员不会亲自攻击您。
- 同样，代码审阅人员也不想以牺牲你雇主的利益为代价来宣传他们雇主的议程。内核开发人员通常希望今后几年能在内核上工作，但他们明白他们的雇主可能会改变。他们真的，几乎毫无例外地，致力于创造他们所能做到的最好的内核；他们并没有试图给雇主的竞争对手造成不适。

所有这些归根结底就是，当审阅者向您发送评论时，您需要注意他们正在进行的技术评论。不要让他们的表达方式或你自己的骄傲阻止此事。当你在一个补丁上得到评论时，花点时间去理解评论人想说什么。如果可能的话，请修复审阅者要求您修复的内容。然后回复审阅者：谢谢他们，并描述你将如何回答他们的问题。

请注意，您不必同意审阅者建议的每个更改。如果您认为审阅者误解了您的代码，请解释到底发生了什么。如果您对建议的更改有技术上的异议，请描述它并证明您对该问题的解决方案是正确的。如果你的解释有道理，审阅者会接受的。不过，如果你的解释证明缺乏说服力，尤其是当其他人开始同意审稿人的观点时，请花些时间重新考虑一下。你很容易对自己解决问题的方法视而不见，以至于你没有意识到某些东西完全是错误的，或者你甚至没有解决正确的问题。

Andrew Morton 建议，每一个不会导致代码更改的审阅评论都应该产生一个额外的代码注释；这可以帮助未来的审阅人员避免第一次出现的问题。

一个致命的错误是忽视评论，希望它们会消失。它们不会走的。如果您在没有对之前收到的评论做出响应的情况下重新发布代码，那么很可能会发现补丁毫无用处。

说到重新发布代码：请记住，审阅者不会记住您上次发布的代码的所有细节。因此，提醒审阅人员以前提出的问题以及您如何处理这些问题总是一个好主意；补丁变更日志是提供此类信息的好地方。审阅者不必搜索列表

档案来熟悉上次所说的内容；如果您帮助他们直接开始，当他们重新查看您的代码时，心情会更好。

如果你已经试着做正确的事情，但事情仍然没有进展呢？大多数技术上的分歧都可以通过讨论来解决，但有时人们仍需要做出决定。如果你真的认为这个决定对你不利，你可以试着向有更高权力的人上诉。对于本文，更高权力的人是 Andrew Morton。Andrew 在内核开发社区中非常受尊敬；他经常为似乎被绝望阻塞的事情清障。尽管如此，不应轻易就直接找 Andrew，也不应在所有其他替代方案都被尝试之前找他。当然，记住，他也可能不同意你的意见。

## 接下来会发生什么

如果一个补丁被认为适合添加到内核中，并且大多数审查问题得到解决，下一步通常是进入子系统维护人员的树中。工作方式因子系统而异；每个维护人员都有自己的工作方式。特别是可能有不止一棵树——也许一棵树专门用于计划下一个合并窗口的补丁，另一棵树用于长期工作。

对于应用到不属于明显子系统树（例如内存管理修补程序）的区域的修补程序，默认树通常上溯到-mm。影响多个子系统的补丁也可以最终进入-mm 树。

包含在子系统树中可以提高补丁的可见性。现在，使用该树的其他开发人员将默认获得补丁。子系统树通常也为 Linux 提供支持，使其内容对整个开发社区可见。在这一点上，您很可能会从一组新的审阅者那里得到更多的评论；这些评论需要像上一轮那样得到回应。

在这时也会发生点什么，这取决于你的补丁的性质，是否与其他人正在做的工作发生冲突。在最坏的情况下，严重的补丁冲突可能会导致一些工作被搁置，以便剩余的补丁可以成形并合并。另一些时候，冲突解决将涉及到与其他开发人员合作，可能还会在树之间移动一些补丁，以确保所有的应用都是干净的。这项工作可能是一件痛苦的事情，但也需庆幸现在的幸福：在 linux-next 树出现之前，这些冲突通常只在合并窗口中出现，必须迅速解决。现在可以在合并窗口打开之前的空闲时间解决这些问题。

有朝一日，如果一切顺利，您将登录并看到您的补丁已经合并到主线内核中。祝贺你！然而，一旦庆祝完了（并且您已经将自己添加到维护人员文件中），就一定要记住一个重要的小事实：工作仍然没有完成。并入主线也带来了它的挑战。

首先，补丁的可见性再次提高。可能会有以前不知道这个补丁的开发者的新一轮评论。忽略它们可能很有诱惑力，因为您的代码不再存在任何被合并的问题。但是，要抵制这种诱惑，您仍然需要对有问题或建议的开发人员作出响应。

不过，更重要的是：将代码包含在主线中会将代码交给更多的一些测试人员。即使您为尚未可用的硬件提供了驱动程序，您也会惊讶于有多少人会将您的代码构建到内核中。当然，如果有测试人员，也可能会有错误报告。

最糟糕的错误报告是回归。如果你的补丁导致回归，你会发现多到让你不舒服的眼睛盯着你；回归需要尽快修复。如果您不愿意或无法修复回归（其他人都不会为您修复），那么在稳定期内，您的补丁几乎肯定会被移除。除了否定您为使补丁进入主线所做的所有工作之外，如果由于未能修复回归而取消补丁，很可能会使将来的工作更难被合并。

在处理完任何回归之后，可能还有其他普通缺陷需要处理。稳定期是修复这些错误并确保代码在主线内核版本中的首次发布尽可能可靠的最好机会。所以，请回应错误报告，并尽可能解决问题。这就是稳定期的目的；一旦解决了旧补丁的任何问题，就可以开始尽情创建新补丁。

别忘了，还有其他节点也可能会创建缺陷报告：下一个主线稳定版本，当著名的发行商选择包含您补丁的内核版本时等等。继续响应这些报告是您工作的基本素养。但是如果这不能提供足够的动机，那么也需要考虑：开发社区会记住那些在合并后对代码失去兴趣的开发人员。下一次你发布补丁时，他们会以你以后不会持续维护它为前提来评估它。

### 其他可能发生的事情

某天，当你打开你的邮件客户端时，看到有人给你寄了一个代码补丁。毕竟，这是让您的代码公开存在的好处之一。如果您同意这个补丁，您可以将它转发给子系统维护人员（确保包含一个正确的 From: 行，这样属性是正确的，并添加一个您自己的 signoff），或者回复一个 Acked-by: 让原始发送者向上发送它。

如果您不同意补丁，请礼貌地回复，解释原因。如果可能的话，告诉作者需要做哪些更改才能让您接受补丁。合并代码的编写者和维护者所反对的补丁的确存在着一定的阻力，但仅此而已。如果你被认为不必要的阻碍了好的工作，那么这些补丁最终会绕过你并进入主线。在 Linux 内核中，没有人对任何代码拥有绝对的否决权。可能除了 Linus。

在非常罕见的情况下，您可能会看到完全不同的东西：另一个开发人员发布了针对您的问题的不同解决方案。在这时，两个补丁之一可能不会被合并，“我的补丁首先发布”不被认为是一个令人信服的技术论据。如果有别人的补丁取代了你的补丁而进入了主线，那么只有一种方法可以回应你：很高兴你的问题解决了，请继续工作吧。以这种方式把某人的工作推到一边可能导致伤心和气馁，但是社区会记住你的反应，即使很久以后他们已经忘记了谁的补丁真正被合并。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:<alexs@kernel.org>)。

---

**Original** Documentation/process/7.AdvancedTopics.rst

**Translator** 时奎亮 Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

校译 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

## 高级主题

现在，希望您能够掌握开发流程的工作方式。然而，还有更多的东西要学！本节将介绍一些主题，这些主题对希望成为 Linux 内核开发过程常规部分的开发人员有帮助。

### 使用 Git 管理补丁

内核使用分布式版本控制始于 2002 年初，当时 Linus 首次开始使用专有的 Bitkeeper 应用程序。虽然 BitKeeper 存在争议，但它所体现的软件版本管理方法却肯定不是。分布式版本控制可以立即加速内核开发项目。现在有好几种免费的 BitKeeper 替代品。但无论好坏，内核项目都已经选择了 Git 作为其工具。

使用 Git 管理补丁可以使开发人员的生活更加轻松，尤其是随着补丁数量的增长。Git 也有其粗糙的边角和一定的危险性，它是一个年轻和强大的工具，仍然在其开发人员完善中。本文档不会试图教会读者如何使用 git；这会是个巨长的文档。相反，这里的重点将是 Git 如何特别适合内核开发过程。想要加快用 Git 速度的开发人员可以在以下网站上找到更多信息：

<https://git-scm.com/>

<https://www.kernel.org/pub/software/scm/git/docs/user-manual.html>

同时网上也能找到各种各样的教程。

在尝试使用它生成补丁供他人使用之前，第一要务是阅读上述网页，对 Git 的工作方式有一个扎实的了解。使用 Git 的开发人员应能进行拉取主线存储库的副本，查询修订历史，提交对树的更改，使用分支等操作。了解 Git 用于重写历史的工具（如 rebase）也很有用。Git 有自己的术语和概念；Git 的新用户应该了解引用、远程分支、索引、快进合并、推拉、游离头等。一开始可能有点吓人，但这些概念不难通过一点学习来理解。

使用 git 生成通过电子邮件提交的补丁是提高速度的一个很好的练习。

当您准备好开始建立 Git 树供其他人查看时，无疑需要一个可以从中拉取的服务器。如果您有一个可以访问因特网的系统，那么使用 git-daemon 设置这样的服务器相对简单。同时，免费的公共托管网站（例如 github）也开始出现在网络上。成熟的开发人员可以在 kernel.org 上获得一个帐户，但这些帐户并不容易得到；更多有关信息，请参阅 <https://kernel.org/faq/>。

正常的 Git 工作流程涉及到许多分支的使用。每一条开发线都可以分为单独的“主题分支”，并独立维护。Git 的分支很容易使用，没有理由不使用它们。而且，在任何情况下，您都不应该在任何您打算让其他人从中拉取的分支中进行开发。应该小心地创建公开可用的分支；当开发分支处于完整状态并已准备好时（而不是之前）才合并开发分支的补丁。

Git 提供了一些强大的工具，可以让您重写开发历史。一个不方便的补丁（比如说，一个打破二分法的补丁，或者有其他一些明显的缺陷）可以在适当的位置修复，或者完全从历史中消失。一个补丁系列可以被重写，就好像它是在今天的主线上写的一样，即使你已经花了几个月的时间在写它。可以透明地将更改从一个分支转移到另一个分支。等等。明智地使用 git 修改历史的能力可以帮助创建问题更少的干净补丁集。

然而，过度使用这种功能可能会导致其他问题，而不仅仅是对创建完美项目历史的简单痴迷。重写历史将重写该历史中包含的更改，将经过测试（希望如此）的内核树变为未经测试的内核树。除此之外，如果开发人员没有共享项目历史，他们就无法轻松地协作；如果您重写了其他开发人员拉入他们存储库的历史，您将使这些开

发人员的生活更加困难。因此，这里有一个简单的经验法则：被导出到其他地方的历史在此后通常被认为是不可变的。

因此，一旦将一组更改推送到公开可用的服务器上，就不应该重写这些更改。如果您尝试强制进行无法快进合并的更改（即不共享同一历史记录的更改），Git 将尝试强制执行此规则。这可能覆盖检查，有时甚至需要重写导出的树。在树之间移动变更集以避免 linux-next 中的冲突就是一个例子。但这种行为应该是罕见的。这就是为什么开发应该在私有分支中进行（必要时可以重写）并且只有在公共分支处于合理的较新状态时才转移到公共分支中的原因之一。

当主线（或其他一组变更所基于的树）前进时，很容易与该树合并以保持领先地位。对于一个私有的分支，rebasing 可能是一个很容易跟上另一棵树的方法，但是一旦一棵树被导出到外界，rebasing 就不可取了。一旦发生这种情况，就必须进行完全合并（merge）。合并有时是很有意义的，但是过于频繁的合并会不必要的扰乱历史。在这种情况下建议的做法是不要频繁合并，通常只在特定的发布点（如主线-rc 发布）合并。如果您对特定的更改感到紧张，则可以始终在私有分支中执行测试合并。在这种情况下，git “rerere” 工具很有用；它能记住合并冲突是如何解决的，这样您就不必重复相同的工作。

关于 Git 这样的工具的一个最大的反复抱怨是：补丁从一个存储库到另一个存储库的大量移动使得很容易陷入错误建议的变更中，这些变更避开审查雷达进入主线。当内核开发人员看到这种情况发生时，他们往往会感到不高兴；在 Git 树上放置未审阅或主题外的补丁可能会影响您将来让树被拉取的能力。引用 Linus 的话：

你可以给我发补丁，但当我从你那里拉取一个 Git 补丁时，我需要知道你清楚自己在做什么，我需要能够相信事情而 \* 无需 \* 手动检查每个单独的更改。

(<http://lwn.net/articles/224135/>)。

为了避免这种情况，请确保给定分支中的所有补丁都与相关主题紧密相关；“驱动程序修复”分支不应更改核心内存管理代码。而且，最重要的是，不要使用 Git 树来绕过审查过程。不时的将树的摘要发布到相关的列表中，在合适时候请求 linux-next 中包含该树。

如果其他人开始发送补丁以包含到您的树中，不要忘记审阅它们。还要确保您维护正确的作者信息；git “am” 工具在这方面做得最好，但是如果补丁通过第三方转发给您，您可能需要在补丁中添加 “From:” 行。

请求拉取时，请务必提供所有相关信息：树的位置、要拉取的分支以及拉取将导致的更改。在这方面 git request-pull 命令非常有用；它将按照其他开发人员所期望的格式化请求，并检查以确保您已记得将这些更改推送到公共服务器。

## 审阅补丁

一些读者显然会反对将本节与“高级主题”放在一起，因为即使是刚开始的内核开发人员也应该审阅补丁。当然，没有比查看其他人发布的代码更好的方法来学习如何在内核环境中编程了。此外，审阅者永远供不应求；通过审阅代码，您可以对整个流程做出重大贡献。

审查代码可能是一副令人生畏的图景，特别是对一个新的内核开发人员来说，他们可能会对公开询问代码感到紧张，而这些代码是由那些有更多经验的人发布的。不过，即使是最有经验的开发人员编写的代码也可以得到改进。也许对（所有）审阅者最好的建议是：把审阅评论当成问题而不是批评。询问“在这条路径中如何释放锁？”总是比说“这里的锁是错误的”更好。

不同的开发人员将从不同的角度审查代码。部分人会主要关注代码风格以及代码行是否有尾随空格。其他人会主要关注补丁作为一个整体实现的变更是否对内核有好处。同时也有人会检查是否存在锁问题、堆栈使用过度、可能的安全问题、在其他地方发现的代码重复、足够的文档、对性能的不利影响、用户空间 ABI 更改等。所有类型的检查，只要它们能引导更好的代码进入内核，都是受欢迎和值得的。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/process/8.Conclusion.rst

**Translator** 时奎亮 Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

校译 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

## 更多信息

关于 Linux 内核开发和相关主题的信息来源很多。首先是在内核源代码分发中找到的文档目录。顶级 [Documentation/translations/zh\\_CN/process/howto.rst](#) 文件是一个重要的起点；[Documentation/translations/zh\\_CN/process/submitting-patches.rst](#) 和 [Documentation/translations/zh\\_CN/process/submitting-drivers.rst](#) 也是所有内核开发人员都应该阅读的内容。许多内部内核 API 都是使用 kerneldoc 机制记录的；“make htmldocs”或“make pdfdocs”可用于以 HTML 或 PDF 格式生成这些文档（尽管某些发行版提供的 tex 版本会遇到内部限制，无法正确处理文档）。

不同的网站在各个细节层次上讨论内核开发。本文作者想谦虚地建议用 <https://lwn.net/> 作为来源；有关许多特定内核主题的信息可以通过以下网址的 LWN 内核索引找到：

<http://lwn.net/kernel/index/>

除此之外，内核开发人员的一个宝贵资源是：

<https://kernelnewbies.org/>

当然，也不应该忘记 <https://kernel.org/>，这是内核发布信息的最终位置。

关于内核开发有很多书：

《Linux 设备驱动程序》第三版 (Jonathan Corbet、Alessandro Rubini 和 Greg Kroah Hartman) 线上版本在 <http://lwn.net/kernel/lDD3/>

《Linux 内核设计与实现》(Robert Love)

《深入理解 Linux 内核》(Daniel Bovet 和 Marco Cesati)

然而，所有这些书都有一个共同的缺点：它们上架时就往往有些过时，而且已经上架一段时间了。不过，在那里还是可以找到相当多的好信息。

有关 git 的文档，请访问：

<https://www.kernel.org/pub/software/scm/git/docs/>

<https://www.kernel.org/pub/software/scm/git/docs/user-manual.html>

## 结论

祝贺所有通过这篇冗长的文档的人。希望它能够帮助您理解 Linux 内核是如何开发的，以及您如何参与这个过程。

最后，重要的是参与。任何开源软件项目都不会超过其贡献者投入其中的总和。Linux 内核的发展速度和以前一样快，因为它得到了大量开发人员的帮助，他们都在努力使它变得更好。内核是一个最成功的例子，说明了当成千上万的人为了一个共同的目标一起工作时，可以做出什么。

不过，内核总是可以从更大的开发人员基础中获益。总有更多的工作要做。但是同样重要的是，Linux 生态系统中的大多数其他参与者可以通过为内核做出贡献而受益。使代码进入主线是提高代码质量、降低维护和分发成本、提高对内核开发方向的影响程度等的关键。这是一种共赢的局面。启动你的编辑器，来加入我们吧；你会非常受欢迎的。

本文档的目的是帮助开发人员（及其经理）以最小的挫折感与开发社区合作。它试图记录这个社区如何以一种不熟悉 Linux 内核开发（或者实际上是自由软件开发）的人可以访问的方式工作。虽然这里有一些技术资料，但这是一个面向过程的讨论，不需要深入了解内核编程就可以理解。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:<alexs@kernel.org>)。

---

### Original Documentation/process/email-clients.rst

译者：

中文版维护者： 贾威威 Harry Wei <[harryxiyou@gmail.com](mailto:harryxiyou@gmail.com)>  
中文版翻译者： 贾威威 Harry Wei <[harryxiyou@gmail.com](mailto:harryxiyou@gmail.com)>  
                  时奎亮 Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>  
中文版校译者： Yinglin Luan <[synmyth@gmail.com](mailto:synmyth@gmail.com)>  
                  Xiaochen Wang <[wangxiaochen0@gmail.com](mailto:wangxiaochen0@gmail.com)>  
                  yaxinsn <[yaxinsn@163.com](mailto:yaxinsn@163.com)>

## Linux 邮件客户端配置信息

### Git

现在大多数开发人员使用 `git send-email` 而不是常规的电子邮件客户端。这方面的手册非常好。在接收端，维护人员使用 `git am` 加载补丁。

如果你是 `git` 新手，那么把你的第一个补丁发送给你自己。将其保存为包含所有标题的原始文本。运行 `git am raw_email.txt`，然后使用 `git log` 查看更改日志。如果工作正常，再将补丁发送到相应的邮件列表。

### 普通配置

Linux 内核补丁是通过邮件被提交的，最好把补丁作为邮件体的内嵌文本。有些维护者接收附件，但是附件的内容格式应该是”`text/plain`”。然而，附件一般是不赞成的，因为这会使补丁的引用部分在评论过程中变得很困难。

用来发送 Linux 内核补丁的邮件客户端在发送补丁时应该处于文本的原始状态。例如，他们不能改变或者删除制表符或者空格，甚至是在每一行的开头或者结尾。

不要通过”`format=flowed`” 模式发送补丁。这样会引起不可预期以及有害的断行。

不要让你的邮件客户端进行自动换行。这样也会破坏你的补丁。

邮件客户端不能改变文本的字符集编码方式。要发送的补丁只能是 ASCII 或者 UTF-8 编码方式，如果你使用 UTF-8 编码方式发送邮件，那么你将会避免一些可能发生的字符集问题。

邮件客户端应该形成并且保持 `References:` 或者 `In-Reply-To:` 标题，那么邮件话题就不会中断。

复制粘帖 (或者剪贴粘帖) 通常不能用于补丁，因为制表符会转换为空格。使用 `xclipboard`, `xclip` 或者 `xcutsel` 也许可以，但是最好测试一下或者避免使用复制粘帖。

不要在使用 PGP/GPG 署名的邮件中包含补丁。这样会使得很多脚本不能读取和适用于你的补丁。(这个问题应该是可以修复的)

在给内核邮件列表发送补丁之前，给自己发送一个补丁是个不错的主意，保存接收到的邮件，将补丁用’`patch`’命令打上，如果成功了，再给内核邮件列表发送。

### 一些邮件客户端提示

这里给出一些详细的 MUA 配置提示，可以用于给 Linux 内核发送补丁。这些并不意味是所有的软件包配置总结。

说明：TUI = 以文本为基础的用户接口 GUI = 图形界面用户接口

### Alpine (TUI)

配置选项：在” Sending Preferences” 部分：

- “Do Not Send Flowed Text” 必须开启
- “Strip Whitespace Before Sending” 必须关闭

当写邮件时，光标应该放在补丁会出现的地方，然后按下 CTRL-R 组合键，使指定的补丁文件嵌入到邮件中。

### Evolution (GUI)

一些开发者成功的使用它发送补丁

当选择邮件选项：**Preformat** 从 Format->Heading->Preformatted (Ctrl-7) 或者工具栏

然后使用： Insert->Text File…(Alt-n x) 插入补丁文件。

你还可以” diff -Nru old.c new.c | xclip”，选择 Preformat，然后使用中间键进行粘贴。

### Kmail (GUI)

一些开发者成功的使用它发送补丁。

默认设置不为 HTML 格式是合适的；不要启用它。

当书写一封邮件的时候，在选项下面不要选择自动换行。唯一的缺点就是你在邮件中输入的任何文本都不会被自动换行，因此你必须在发送补丁之前手动换行。最简单的方法就是启用自动换行来书写邮件，然后把它保存为草稿。一旦你在草稿中再次打开它，它已经全部自动换行了，那么你的邮件虽然没有选择自动换行，但是还不会失去已有的自动换行。

在邮件的底部，插入补丁之前，放上常用的补丁定界符：三个连字号（—）。

然后在” Message” 菜单条目，选择插入文件，接着选取你的补丁文件。还有一个额外的选项，你可以通过它配置你的邮件建立工具栏菜单，还可以带上” insert file” 图标。

你可以安全地通过 GPG 标记附件，但是内嵌补丁最好不要使用 GPG 标记它们。作为内嵌文本的签发补丁，当从 GPG 中提取 7 位编码时会使他们变的更加复杂。

如果你非要以附件的形式发送补丁，那么就右键点击附件，然后选中属性，突出” Suggest automatic display”，这样内嵌附件更容易让读者看到。

当你要保存将要发送的内嵌文本补丁，你可以从消息列表窗格选择包含补丁的邮件，然后右击选择“save as”。你可以使用一个没有更改的包含补丁的邮件，如果它是以正确的形式组成。当你正真在它自己的窗口之下察看，那时没有选项可以保存邮件-已经有一个这样的 bug 被汇报到了 kmail 的 bugzilla 并且希望这将会被处理。邮件是以只针对某个用户可读写的权限被保存的，所以如果你想把邮件复制到其他地方，你不得不把他们的权限改为组或者整体可读。

## Lotus Notes (GUI)

不要使用它。

## Mutt (TUI)

很多 Linux 开发人员使用 mutt 客户端，所以证明它肯定工作的非常漂亮。

Mutt 不自带编辑器，所以不管你使用什么编辑器都不应该带有自动断行。大多数编辑器都带有一个”insert file” 选项，它可以通过不改变文件内容的方式插入文件。

‘vim’ 作为 mutt 的编辑器: set editor=“vi”

如果使用 xclip，敲入以下命令:set paste 按中键之前或者 shift-insert 或者使用:r filename 如果想要把补丁作为内嵌文本。(a)ttach 工作的很好，不带有”set paste”。

你可以通过 git format-patch 生成补丁，然后用 Mutt 发送它们:

```
$ mutt -H 0001-some-bug-fix.patch
```

配置选项: 它应该以默认设置的形式工作。然而，把”send\_charset” 设置为”us-ascii::utf-8” 也是一个不错的主意。

Mutt 是高度可配置的。这里是个使用 mutt 通过 Gmail 发送的补丁的最小配置:

```
# .muttrc
# ====== IMAP ======
set imap_user = 'yourusername@gmail.com'
set imap_pass = 'yourpassword'
set spoolfile = imaps://imap.gmail.com/INBOX
set folder = imaps://imap.gmail.com/
set record="imaps://imap.gmail.com/[Gmail]/Sent Mail"
set postponed="imaps://imap.gmail.com/[Gmail]/Drafts"
set mbox="imaps://imap.gmail.com/[Gmail]/All Mail"

# ====== SMTP ======
set smtp_url = "smtp://username@smtp.gmail.com:587/"
set smtp_pass = $imap_pass
set ssl_force_tls = yes # Require encrypted connection

# ====== Composition ======
set editor = `echo \$EDITOR`
set edit_headers = yes # See the headers when editing
set charset = UTF-8      # value of $LANG; also fallback for send_charset
# Sender, email address, and sign-off line must match
unset use_domain        # because joe@localhost is just embarrassing
set realname = "YOUR NAME"
set from = "username@gmail.com"
set use_from = yes
```

Mutt 文档含有更多信息：

<http://dev.mutt.org/trac/wiki/UseCases/Gmail>

<http://dev.mutt.org/doc/manual.html>

### Pine (TUI)

Pine 过去有一些空格删减问题，但是这些现在应该都被修复了。

如果可以，请使用 alpine(pine 的继承者)

配置选项： - 最近的版本需要消除流程文本 - “no-strip-whitespace-before-send” 选项也是需要的。

### Sylpheed (GUI)

- 内嵌文本可以很好的工作（或者使用附件）。
- 允许使用外部的编辑器。
- 对于目录较多时非常慢。
- 如果通过 non-SSL 连接，无法使用 TLS SMTP 授权。
- 在组成窗口中有一个很有用的 ruler bar。
- 给地址本中添加地址就不会正确的了解显示名。

### Thunderbird (GUI)

默认情况下，thunderbird 很容易损坏文本，但是还有一些方法可以强制它变得更好。

- 在用户帐号设置里，组成和寻址，不要选择“Compose messages in HTML format”。
- 编辑你的 Thunderbird 配置设置来使它不要拆行使用：user\_pref("mailnews.wraplength", 0);
- 编辑你的 Thunderbird 配置设置，使它不要使用“format=flowed”格式：user\_pref("mailnews.send\_plaintext\_flowed", false);
- 你需要使 Thunderbird 变为预先格式方式：如果默认情况下你书写的是 HTML 格式，那不是很难。仅仅从标题栏的下拉框中选择“Preformat”格式。如果默认情况下你书写的是文本格式，你不得把它改为 HTML 格式（仅仅作为一次性的）来书写新的消息，然后强制使它回到文本格式，否则它就会拆行。要实现它，在写信的图标上使用 shift 键来使它变为 HTML 格式，然后标题栏的下拉框中选择“Preformat”格式。
- 允许使用外部的编辑器：针对 Thunderbird 打补丁最简单的方法就是使用一个“external editor”扩展，然后使用你最喜欢的 \$EDITOR 来读取或者合并补丁到文本中。要实现它，可以下载并且安装这个扩展，然后添加一个使用它的按键 View->Toolbars->Customize…最后当你书写信息的时候仅仅点击它就可以了。

## TkRat (GUI)

可以使用它。使用”Insert file...”或者外部的编辑器。

## Gmail (Web GUI)

不要使用它发送补丁。

Gmail 网页客户端自动地把制表符转换为空格。

虽然制表符转换为空格问题可以被外部编辑器解决，同时它还会使用回车换行把每行拆分为 78 个字符。

另一个问题是 Gmail 还会把任何不是 ASCII 的字符的信息改为 base64 编码。它把东西变的像欧洲人的名字。

###

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/process/license-rules.rst

**Translator** Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

## Linux 内核许可规则

Linux 内核根据 LICENSES/pREFERRED/GPL-2.0 中提供的 GNU 通用公共许可证版本 2 (GPL-2.0) 的条款提供，并在 LICENSES/EXCEPTIONS/Linux-syscall-note 中显式描述了例外的系统调用，如 COPYING 文件中所述。

此文档文件提供了如何对每个源文件进行注释以使其许可证清晰明确的说明。它不会取代内核的许可证。

内核源代码作为一个整体适用于 COPYING 文件中描述的许可证，但是单个源文件可以具有不同的与 GPL-2.0 兼容的许可证：

GPL-1.0+ : GNU 通用公共许可证 v1.0 或更高版本
GPL-2.0+ : GNU 通用公共许可证 v2.0 或更高版本
LGPL-2.0 : 仅限 GNU 库通用公共许可证 v2
LGPL-2.0+ : GNU 库通用公共许可证 v2 或更高版本
LGPL-2.1 : 仅限 GNU 宽通用公共许可证 v2.1
LGPL-2.1+ : GNU 宽通用公共许可证 v2.1 或更高版本

除此之外，个人文件可以在双重许可下提供，例如一个兼容的 GPL 变体，或者 BSD，MIT 等许可。

用户空间 API (UAPI) 头文件描述了用户空间程序与内核的接口，这是一种特殊情况。根据内核 COPYING 文件中的注释，syscall 接口是一个明确的边界，它不会将 GPL 要求扩展到任何使用它与内核通信的软件。由于 UAPI 头文件必须包含在创建在 Linux 内核上运行的可执行文件的任何源文件中，因此此例外必须记录在特别的许可证表述中。

表达源文件许可证的常用方法是将匹配的样板文本添加到文件的顶部注释中。由于格式，拼写错误等，这些“样板”很难通过那些在上下文中使用的验证许可证合规性的工具。

样板文本的替代方法是在每个源文件中使用软件包数据交换 (SPDX) 许可证标识符。SPDX 许可证标识符是机器可解析的，并且是用于提供文件内容的许可证的精确缩写。SPDX 许可证标识符由 Linux 基金会的 SPDX 工作组管理，并得到了整个行业，工具供应商和法律团队的合作伙伴的一致同意。有关详细信息，请参阅 <https://spdx.org/>

Linux 内核需要所有源文件中的精确 SPDX 标识符。内核中使用的有效标识符在[许可证标识符](#)一节中进行了解释，并且已可以在 <https://spdx.org/licenses/> 上的官方 SPDX 许可证列表中检索，并附带许可证文本。

### 许可标识符语法

#### 1. 安置：

内核文件中的 SPDX 许可证标识符应添加到可包含注释的文件中的第一行。对于大多数文件，这是第一行，除了那些在第一行中需要’#!PATH\_TO\_INTERPRETER’的脚本。对于这些脚本，SPDX 标识符进入第二行。

#### 2. 风格：

SPDX 许可证标识符以注释的形式添加。注释样式取决于文件类型：

```
C source: // SPDX-License-Identifier: <SPDX License Expression>
C header: /* SPDX-License-Identifier: <SPDX License Expression> */
ASM:      /* SPDX-License-Identifier: <SPDX License Expression> */
scripts:  # SPDX-License-Identifier: <SPDX License Expression>
.rst:     .. SPDX-License-Identifier: <SPDX License Expression>
.dts{i}: // SPDX-License-Identifier: <SPDX License Expression>
```

如果特定工具无法处理标准注释样式，则应使用工具接受的相应注释机制。这是在 C 头文件中使用 “`/**/`” 样式注释的原因。过去在使用生成的.lds 文件中观察到构建被破坏，其中’ld’ 无法解析 C++ 注释。现在已经解决了这个问题，但仍然有较旧的汇编程序工具无法处理 C++ 样式的注释。

### 3. 句法:

< SPDX 许可证表达式 > 是 SPDX 许可证列表中的 SPDX 短格式许可证标识符，或者在许可证例外适用时由“WITH”分隔的两个 SPDX 短格式许可证标识符的组合。当应用多个许可证时，表达式由分隔子表达式的关键字“AND”，“OR”组成，并由“(，)”包围。

带有“或更高”选项的 [L]GPL 等许可证的许可证标识符通过使用“+”来表示“或更高”选项来构建。：

```
// SPDX-License-Identifier: GPL-2.0+
// SPDX-License-Identifier: LGPL-2.1+
```

当需要修正的许可证时，应使用 WITH。例如，linux 内核 UAPI 文件使用表达式：

```
// SPDX-License-Identifier: GPL-2.0 WITH Linux-syscall-note
// SPDX-License-Identifier: GPL-2.0+ WITH Linux-syscall-note
```

其它在内核中使用 WITH 例外的事例如下：

```
// SPDX-License-Identifier: GPL-2.0 WITH mif-exception
// SPDX-License-Identifier: GPL-2.0+ WITH GCC-exception-2.0
```

例外只能与特定的许可证标识符一起使用。有效的许可证标识符列在异常文本文件的标记中。有关详细信息，请参阅[许可标识符](#)一章中的[例外](#)。

如果文件是双重许可且只选择一个许可证，则应使用 OR。例如，一些 dtsi 文件在双许可下可用：

```
// SPDX-License-Identifier: GPL-2.0 OR BSD-3-Clause
```

内核中双许可文件中许可表达式的示例：

```
// SPDX-License-Identifier: GPL-2.0 OR MIT
// SPDX-License-Identifier: GPL-2.0 OR BSD-2-Clause
// SPDX-License-Identifier: GPL-2.0 OR Apache-2.0
// SPDX-License-Identifier: GPL-2.0 OR MPL-1.1
// SPDX-License-Identifier: (GPL-2.0 WITH Linux-syscall-note) OR MIT
// SPDX-License-Identifier: GPL-1.0+ OR BSD-3-Clause OR OpenSSL
```

如果文件具有多个许可证，其条款全部适用于使用该文件，则应使用 AND。例如，如果代码是从另一个项目继承的，并且已经授予了将其放入内核的权限，但原始许可条款需要保持有效：

```
// SPDX-License-Identifier: (GPL-2.0 WITH Linux-syscall-note) AND MIT
```

另一个需要遵守两套许可条款的例子是：

```
// SPDX-License-Identifier: GPL-1.0+ AND LGPL-2.1+
```

### 许可标识符

当前使用的许可证以及添加到内核的代码许可证可以分解为：

#### 1. 优先许可:

应尽可能使用这些许可证，因为它们已知完全兼容并广泛使用。这些许可证在内核目录：

```
LICENSES/preferred/
```

此目录中的文件包含完整的许可证文本和元标记。文件名与 SPDX 许可证标识符相同，后者应用于源文件中的许可证。

例如：

```
LICENSES/preferred/GPL-2.0
```

包含 GPLv2 许可证文本和所需的元标签：

```
LICENSES/preferred/MIT
```

包含 MIT 许可证文本和所需的元标记

元标记：

许可证文件中必须包含以下元标记：

- Valid-License-Identifier:

一行或多行，声明那些许可标识符在项目内有效，以引用此特定许可的文本。通常这是一个有效的标识符，但是例如对于带有‘或更高’选项的许可证，两个标识符都有效。

- SPDX-URL:

SPDX 页面的 URL，其中包含与许可证相关的其他信息。

- Usage-Guidance:

使用建议的自由格式文本。该文本必须包含 SPDX 许可证标识符的正确示例，因为它们应根据 [许可证语法](#) 指南放入源文件中。

- License-Text:

此标记之后的所有文本都被视为原始许可文本

文件格式示例：

```
Valid-License-Identifier: GPL-2.0
Valid-License-Identifier: GPL-2.0+
SPDX-URL: https://spdx.org/licenses/GPL-2.0.html
Usage-Guide:
To use this license in source code, put one of the following SPDX
tag/value pairs into a comment according to the placement
```

guidelines in the licensing rules documentation.

For 'GNU General Public License (GPL) version 2 only' use:

SPDX-License-Identifier: GPL-2.0

For 'GNU General Public License (GPL) version 2 or any later version' use:

SPDX-License-Identifier: GPL-2.0+

License-Text:

Full license text

SPDX-License-Identifier: MIT

SPDX-URL: <https://spdx.org/licenses/MIT.html>

Usage-Guide:

To use this license in source code, put the following SPDX tag/value pair into a comment according to the placement guidelines in the licensing rules documentation.

SPDX-License-Identifier: MIT

License-Text:

Full license text

## 2. 不推荐的许可证:

这些许可证只应用于现有代码或从其他项目导入代码。这些许可证在内核目录:

LICENSES/other/

此目录中的文件包含完整的许可证文本和元标记。文件名与 SPDX 许可证标识符相同，后者应用于源文件中的许可证。

例如:

LICENSES/other/ISC

包含国际系统联合许可文本和所需的元标签:

LICENSES/other/ZLib

包含 ZLIB 许可文本和所需的元标签.

元标签:

“其他”许可证的元标签要求与优先许可 的要求相同。

文件格式示例:

Valid-License-Identifier: ISC

SPDX-URL: <https://spdx.org/licenses/ISC.html>

Usage-Guide:

Usage of this license in the kernel for new code is discouraged and it should solely be used for importing code from an already existing project.

To use this license in source code, put the following SPDX tag/value pair into a comment according to the placement guidelines in the licensing rules documentation.

SPDX-License-Identifier: ISC

License-Text:

Full license text

### 3. 例外:

某些许可证可以修改，并允许原始许可证不具有的某些例外权利。这些例外在内核目录:

LICENSES/exceptions/

此目录中的文件包含完整的例外文本和所需的[例外元标记](#)。

例如:

LICENSES/exceptions/Linux-syscall-note

包含 Linux 内核的 COPYING 文件中记录的 Linux 系统调用例外，该文件用于 UAPI 头文件。例如:

LICENSES/exceptions/GCC-exception-2.0

包含 GCC' 链接例外'，它允许独立于其许可证的任何二进制文件与标记有此例外的文件的编译版本链接。这是从 GPL 不兼容源代码创建可运行的可执行文件所必需的。

例外元标记:

以下元标记必须在例外文件中可用:

- SPDX-Exception-Identifier:

一个可与 SPDX 许可证标识符一起使用的例外标识符。

- SPDX-URL:

SPDX 页面的 URL，其中包含与例外相关的其他信息。

- SPDX-Licenses:

以逗号分隔的例外可用的 SPDX 许可证标识符列表。

- Usage-Guidance:

使用建议的自由格式文本。必须在文本后面加上 SPDX 许可证标识符的正确示例，因为它们应根据[许可证标识符语法](#)指南放入源文件中。

- Exception-Text:

此标记之后的所有文本都被视为原始异常文本

文件格式示例:

```
SPDX-Exception-Identifier: Linux-syscall-note
SPDX-URL: https://spdx.org/licenses/Linux-syscall-note.html
SPDX-Licenses: GPL-2.0, GPL-2.0+, GPL-1.0+, LGPL-2.0, LGPL-2.0+, LGPL-2.1, LGPL-2.1+
Usage-Guidance:
This exception is used together with one of the above SPDX-Licenses
to mark user-space API (uapi) header files so they can be included
into non GPL compliant user-space application code.
To use this exception add it with the keyword WITH to one of the
identifiers in the SPDX-Licenses tag:
SPDX-License-Identifier: <SPDX-License> WITH Linux-syscall-note
Exception-Text:
Full exception text
```

```
SPDX-Exception-Identifier: GCC-exception-2.0
SPDX-URL: https://spdx.org/licenses/GCC-exception-2.0.html
SPDX-Licenses: GPL-2.0, GPL-2.0+
Usage-Guidance:
The "GCC Runtime Library exception 2.0" is used together with one
of the above SPDX-Licenses for code imported from the GCC runtime
library.
To use this exception add it with the keyword WITH to one of the
identifiers in the SPDX-Licenses tag:
SPDX-License-Identifier: <SPDX-License> WITH GCC-exception-2.0
Exception-Text:
Full exception text
```

所有 SPDX 许可证标识符和例外都必须在 LICENSES 子目录中具有相应的文件。这是允许工具验证（例如 `checkpatch.pl`）以及准备好从源读取和提取许可证所必需的，这是各种 FOSS 组织推荐的，例如 [FSFE REUSE initiative](#)。

## 模块许可

可加载内核模块还需要 MODULE\_LICENSE () 标记。此标记既不替代正确的源代码许可证信息（SPDX-License-Identifier），也不以任何方式表示或确定提供模块源代码的确切许可证。

此标记的唯一目的是提供足够的信息，该模块是否是自由软件或者是内核模块加载器和用户空间工具的专有模块。

MODULE\_LICENSE () 的有效许可证字符串是:

“GPL”	模块是根据 GPL 版本 2 许可的。这并不表示仅限于 GPL-2.0 或 GPL-2.0 或更高版本之间的任何区别。最正确许可证信息只能通过相应源文件中的许可证信息来确定
“GPL v2”	和“GPL”相同，它的存在是因为历史原因。
“GPL and additional rights”	表示模块源在 GPL v2 变体和 MIT 许可下双重许可的历史变体。请不要在新代码中使用。
“Dual MIT/GPL”	表达该模块在 GPL v2 变体或 MIT 许可证选择下双重许可的正确方式。
“Dual BSD/GPL”	该模块根据 GPL v2 变体或 BSD 许可证选择进行双重许可。BSD 许可证的确切变体只能通过相应源文件中的许可证信息来确定。
“Dual MPL/GPL”	该模块根据 GPL v2 变体或 Mozilla Public License (MPL) 选项进行双重许可。MPL 许可证的确切变体只能通过相应的源文件中的许可证信息来确定。
“Proprietary”	该模块属于专有许可。此字符串仅用于专有的第三方模块，不能用于在内核树中具有源代码的模块。以这种方式标记的模块在加载时会使用‘P’标记污染内核，并且内核模块加载器拒绝将这些模块链接到使用 EXPORT_SYMBOL_GPL() 导出的符号。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/process/kernel-enforcement-statement.rst

**Translator** Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

## Linux 内核执行声明

作为 Linux 内核的开发人员，我们对如何使用我们的软件以及如何实施软件许可证有着浓厚的兴趣。遵守 GPL-2.0 的互惠共享义务对我们软件和社区的长期可持续性至关重要。

虽然有权强制执行对我们社区的贡献中的单独版权权益，但我们有共同的利益，即确保个人强制执行行动的方式有利于我们的社区，不会对我们软件生态系统的健康和增长产生意外的负面影响。为了阻止无益的执法行动，我们同意代表我们自己和我们版权利益的任何继承人对 Linux 内核用户作出以下符合我们开发社区最大利益的承诺：

尽管有 GPL-2.0 的终止条款，我们同意，采用以下 GPL-3.0 条款作为我们许可证下的附加许可，作为任何对许可证下权利的非防御性主张，这符合我们开发社区的最佳利益。

但是，如果您停止所有违反本许可证的行为，则您从特定版权持有人处获得的许可证将被恢复：(a) 暂时恢复，除非版权持有人明确并最终终止您的许可证；以及 (b) 永久恢复，如果版权持有人未能在你终止违反后 60 天内以合理方式通知您违反本许可证的行为，则永久恢复您的许可证。

此外，如果版权所有者以某种合理的方式通知您违反了本许可，这是您第一次从该版权所有者处收到违反本许可的通知（对于任何作品），并且您在收到通知后的 30 天内纠正违规行为。则您从特定版权所有者处获得的许可将永久恢复。

我们提供这些保证的目的是鼓励更多地使用该软件。我们希望公司和个人使用、修改和分发此软件。我们希望以公开和透明的方式与用户合作，以消除我们对法规遵从性或强制执行的任何不确定性，这些不确定性可能会限制我们软件的采用。我们将法律行动视为最后手段，只有在其他社区努力未能解决这一问题时才采取行动。

最后，一旦一个不合规问题得到解决，我们希望用户会感到欢迎，加入我们为之努力的这个项目。共同努力，我们会更强大。

除了下面提到的以外，我们只为自己说话，而不是为今天、过去或将来可能为之工作的任何公司说话。

- Laura Abbott
- Bjorn Andersson (Linaro)
- Andrea Arcangeli
- Neil Armstrong
- Jens Axboe
- Pablo Neira Ayuso
- Khalid Aziz
- Ralf Baechle
- Felipe Balbi
- Arnd Bergmann
- Ard Biesheuvel
- Tim Bird

- Paolo Bonzini
- Christian Borntraeger
- Mark Brown (Linaro)
- Paul Burton
- Javier Martinez Canillas
- Rob Clark
- Kees Cook (Google)
- Jonathan Corbet
- Dennis Dalessandro
- Vivien Didelot (Savoir-faire Linux)
- Hans de Goede
- Mel Gorman (SUSE)
- Sven Eckelmann
- Alex Elder (Linaro)
- Fabio Estevam
- Larry Finger
- Bhumika Goyal
- Andy Gross
- Juergen Gross
- Shawn Guo
- Ulf Hansson
- Stephen Hemminger (Microsoft)
- Tejun Heo
- Rob Herring
- Masami Hiramatsu
- Michal Hocko
- Simon Hormann
- Johan Hovold (Hovold Consulting AB)
- Christophe JAILLET
- Olof Johansson

- Lee Jones (Linaro)
- Heiner Kallweit
- Srinivas Kandagatla
- Jan Kara
- Shuah Khan (Samsung)
- David Kershner
- Jaegeuk Kim
- Namhyung Kim
- Colin Ian King
- Jeff Kirsher
- Greg Kroah-Hartman (Linux Foundation)
- Christian König
- Vinod Koul
- Krzysztof Kozlowski
- Viresh Kumar
- Aneesh Kumar K.V
- Julia Lawall
- Doug Ledford
- Chuck Lever (Oracle)
- Daniel Lezcano
- Shaohua Li
- Xin Long
- Tony Luck
- Catalin Marinas (Arm Ltd)
- Mike Marshall
- Chris Mason
- Paul E. McKenney
- Arnaldo Carvalho de Melo
- David S. Miller
- Ingo Molnar

- Kuninori Morimoto
- Trond Myklebust
- Martin K. Petersen (Oracle)
- Borislav Petkov
- Jiri Pirko
- Josh Poimboeuf
- Sebastian Reichel (Collabora)
- Guenter Roeck
- Joerg Roedel
- Leon Romanovsky
- Steven Rostedt (VMware)
- Frank Rowand
- Ivan Safonov
- Anna Schumaker
- Jes Sorensen
- K.Y. Srinivasan
- David Sterba (SUSE)
- Heiko Stuebner
- Jiri Kosina (SUSE)
- Willy Tarreau
- Dmitry Torokhov
- Linus Torvalds
- Thierry Reding
- Rik van Riel
- Luis R. Rodriguez
- Geert Uytterhoeven (Glider bvba)
- Eduardo Valentin (Amazon.com)
- Daniel Vetter
- Linus Walleij
- Richard Weinberger

- Dan Williams
- Rafael J. Wysocki
- Arvind Yadav
- Masahiro Yamada
- Wei Yongjun
- Lv Zheng
- Marc Zyngier (Arm Ltd)

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/process/kernel-driver-statement.rst

**Translator** Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

## 内核驱动声明

### 关于 Linux 内核模块的立场声明

我们，以下署名的 Linux 内核开发人员，认为任何封闭源 Linux 内核模块或驱动程序都是有害的和不可取的。我们已经一再发现它们对 Linux 用户，企业和更大的 Linux 生态系统有害。这样的模块否定了 Linux 开发模型的开放性，稳定性，灵活性和可维护性，并使他们的用户无法使用 Linux 社区的专业知识。提供闭源内核模块的供应商迫使其客户放弃 Linux 的主要优势或选择新的供应商。因此，为了充分利用开源所提供的成本节省和共享支持优势，我们敦促供应商采取措施，以开源内核代码在 Linux 上为其客户提供支持。

我们只为自己说话，而不是我们今天可能会为之工作，过去或将来会为之工作的任何公司。

- Dave Airlie
- Nick Andrew
- Jens Axboe
- Ralf Baechle
- Felipe Balbi
- Ohad Ben-Cohen

- Muli Ben-Yehuda
- Jiri Benc
- Arnd Bergmann
- Thomas Bogendoerfer
- Vitaly Bordug
- James Bottomley
- Josh Boyer
- Neil Brown
- Mark Brown
- David Brownell
- Michael Buesch
- Franck Bui-Huu
- Adrian Bunk
- François Cami
- Ralph Campbell
- Luiz Fernando N. Capitulino
- Mauro Carvalho Chehab
- Denis Cheng
- Jonathan Corbet
- Glauber Costa
- Alan Cox
- Magnus Damm
- Ahmed S. Darwish
- Robert P. J. Day
- Hans de Goede
- Arnaldo Carvalho de Melo
- Helge Deller
- Jean Delvare
- Mathieu Desnoyers
- Sven-Thorsten Dietrich

- Alexey Dobriyan
- Daniel Drake
- Alex Dubov
- Randy Dunlap
- Michael Ellerman
- Pekka Enberg
- Jan Engelhardt
- Mark Fasheh
- J. Bruce Fields
- Larry Finger
- Jeremy Fitzhardinge
- Mike Frysinger
- Kumar Gala
- Robin Getz
- Liam Girdwood
- Jan-Benedict Glaw
- Thomas Gleixner
- Brice Goglin
- Cyrill Gorcunov
- Andy Gospodarek
- Thomas Graf
- Krzysztof Halasa
- Harvey Harrison
- Stephen Hemminger
- Michael Hennerich
- Tejun Heo
- Benjamin Herrenschmidt
- Kristian Høgsberg
- Henrique de Moraes Holschuh
- Marcel Holtmann

- Mike Isely
- Takashi Iwai
- Olof Johansson
- Dave Jones
- Jesper Juhl
- Matthias Kaehlcke
- Kenji Kaneshige
- Jan Kara
- Jeremy Kerr
- Russell King
- Olaf Kirch
- Roel Kluin
- Hans-Jürgen Koch
- Auke Kok
- Peter Korsgaard
- Jiri Kosina
- Aaro Koskinen
- Mariusz Kozlowski
- Greg Kroah-Hartman
- Michael Krufky
- Aneesh Kumar
- Clemens Ladisch
- Christoph Lameter
- Gunnar Larisch
- Anders Larsen
- Grant Likely
- John W. Linville
- Yinghai Lu
- Tony Luck
- Pavel Machek

- Matt Mackall
- Paul Mackerras
- Roland McGrath
- Patrick McHardy
- Kyle McMarn
- Paul Menage
- Thierry Merle
- Eric Miao
- Akinobu Mita
- Ingo Molnar
- James Morris
- Andrew Morton
- Paul Mundt
- Oleg Nesterov
- Luca Olivetti
- S.Çağlar Onur
- Pierre Ossman
- Keith Owens
- Venkatesh Pallipadi
- Nick Pigglin
- Nicolas Pitre
- Evgeniy Polyakov
- Richard Purdie
- Mike Rapoport
- Sam Ravnborg
- Gerrit Renker
- Stefan Richter
- David Rientjes
- Luis R. Rodriguez
- Stefan Roes

- Francois Romieu
- Rami Rosen
- Stephen Rothwell
- Maciej W. Rozycki
- Mark Salyzyn
- Yoshinori Sato
- Deepak Saxena
- Holger Schurig
- Amit Shah
- Yoshihiro Shimoda
- Sergei Shtylyov
- Kay Sievers
- Sebastian Siewior
- Rik Snel
- Jes Sorensen
- Alexey Starikovskiy
- Alan Stern
- Timur Tabi
- Hirokazu Takata
- Eliezer Tamir
- Eugene Teo
- Doug Thompson
- FUJITA Tomonori
- Dmitry Torokhov
- Marcelo Tosatti
- Steven Toth
- Theodore Tso
- Matthias Urlichhs
- Geert Uytterhoeven
- Arjan van de Ven

- Ivo van Doorn
- Rik van Riel
- Wim Van Sebroeck
- Hans Verkuil
- Horst H. von Brand
- Dmitri Vorobiev
- Anton Vorontsov
- Daniel Walker
- Johannes Weiner
- Harald Welte
- Matthew Wilcox
- Dan J. Williams
- Darrick J. Wong
- David Woodhouse
- Chris Wright
- Bryan Wu
- Rafael J. Wysocki
- Herbert Xu
- Vlad Yasevich
- Peter Zijlstra
- Bartłomiej Zolnierkiewicz

其它大多数开发人员感兴趣的社区指南：

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/process/submitting-drivers.rst

如果想评论或更新本文的内容, 请直接联系原文档的维护者。如果你使用英文交流有困难的话, 也可以向中文版维护者求助。如果本翻译更新不及时或者翻译存在问题, 请联系中文版维护者:

中文版维护者: 李阳 Li Yang <leoyang.li@nxp.com>  
中文版翻译者: 李阳 Li Yang <leoyang.li@nxp.com>  
中文版校译者: 陈琦 Maggie Chen <chenqi@beyondsoft.com>  
王聪 Wang Cong <xiyou.wangcong@gmail.com>  
张巍 Zhang Wei <wezhang@outlook.com>

## 如何向 Linux 内核提交驱动程序

这篇文档将会解释如何向不同的内核源码树提交设备驱动程序。请注意, 如果你感兴趣的是显卡驱动程序, 你也许应该访问 XFree86 项目 (<https://www.xfree86.org/>) 和／或 X.org 项目 (<https://x.org>)。

另请参阅[如何让你的改动进入内核](#) 文档。

## 分配设备号

块设备和字符设备的主设备号与从设备号是由 Linux 命名编号分配权威 LANANA (现在是 Torben Mathiasen) 负责分配。申请的网址是 <https://www.lanana.org/>。即使不准备提交到主流内核的设备驱动也需要在这里分配设备号。有关详细信息, 请参阅 Documentation/admin-guide/devices.rst。

如果你使用的不是已经分配的设备号, 那么当你提交设备驱动的时候, 它将会被强制分配一个新的设备号, 即便这个设备号和你之前发给客户的截然不同。

## 设备驱动的提交对象

**Linux 2.0:** 此内核源码树不接受新的驱动程序。

**Linux 2.2:** 此内核源码树不接受新的驱动程序。

**Linux 2.4:** 如果所属的代码领域在内核的 MAINTAINERS 文件中列有一个总维护者, 那么请将驱动程序提交给他。如果此维护者没有回应或者你找不到恰当的维护者, 那么请联系 Willy Tarreau <[w@1wt.eu](mailto:w@1wt.eu)>。

**Linux 2.6:** 除了遵循和 2.4 版内核同样的规则外, 你还需要在 linux-kernel 邮件列表上跟踪最新的 API 变化。向 Linux 2.6 内核提交驱动的顶级联系人是 Andrew Morton <[akpm@linux-foundation.org](mailto:akpm@linux-foundation.org)>。

## 决定设备驱动能否被接受的条件

**许可:** 代码必须使用 **GNU 通用公开许可证 (GPL)** 提交给 **Linux**, 但是 我们并不要求 GPL 是唯一的许可。你或许会希望同时使用多种许可证发布, 如果希望驱动程序可以被其他开源社区 (比如 BSD) 使用。请参考 include/linux/module.h 文件中所列出的可被接受共存的许可。

**版权:** 版权所有者必须同意使用 **GPL** 许可。最好提交者和版权所有者 是相同个人或实体。否则, 必需列出授权使用 GPL 的版权所有人或实体, 以备验证之需。

**接口：**如果你的驱动程序使用现成的接口并且和其他同类的驱动程序行为相似，而不是去发明无谓的新接口，那么它将会更容易被接受。如果你需要一个 Linux 和 NT 的通用驱动接口，那么请在用户空间实现它。

**代码：**请使用 `Documentation/process/coding-style.rst` 中所描述的 **Linux 代码风格**。如果你的某些代码段（例如那些与 Windows 驱动程序包共享的代码段）需要使用其他格式，而你却只希望维护一份代码，那么请将它们很好地区分出来，并且注明原因。

**可移植性：**请注意，指针并不永远是 32 位的，不是所有的计算机都使用小尾模式 (little endian) 存储数据，不是所有的人都拥有浮点单元，不要随便在你的驱动程序里嵌入 x86 汇编指令。只能在 x86 上运行的驱动程序一般是不受欢迎的。虽然你可能只有 x86 硬件，很难测试驱动程序在其他平台上是否可用，但是确保代码可以被轻松地移植却是很简单的。

**清晰度：**做到所有人都能修补这个驱动程序将会很有好处，因为这样你将会直接收到修复的补丁而不是 bug 报告。如果你提交一个试图隐藏硬件工作机理的驱动程序，那么它将会被扔进废纸篓。

**电源管理：**因为 Linux 正在被很多移动设备和桌面系统使用，所以你的驱动程序也很有可能被使用在这些设备上。它应该支持最基本的电源管理，即在需要的情况下实现系统级休眠和唤醒要用到的 `.suspend` 和 `.resume` 函数。你应该检查你的驱动程序是否能正确地处理休眠与唤醒，如果实在无法确认，请至少把 `.suspend` 函数定义成返回 -ENOSYS (功能未实现) 错误。你还应该尝试确保你的驱动在什么都不干的情况下将耗电降到最低。要获得驱动程序测试的指导，请参阅 `Documentation/power/drivers-testing.rst`。有关驱动程序电源管理问题相对全面的概述，请参阅 `Documentation/driver-api/pm/devices.rst`。

**管理：**如果一个驱动程序的作者还在进行有效的维护，那么通常除了那些明显正确且不需要任何检查的补丁以外，其他所有的补丁都会被转发给作者。如果你希望成为驱动程序的联系人和更新者，最好在代码注释中写明并且在 `MAINTAINERS` 文件中加入这个驱动程序的条目。

## 不影响设备驱动能否被接受的条件

**供应商：**由硬件供应商来维护驱动程序通常是一件好事。不过，如果源码树里已经有其他人提供了可稳定工作的驱动程序，那么请不要期望“我是供应商”会成为内核改用你的驱动程序的理由。理想的情况是：供应商与现有驱动程序的作者合作，构建一个统一完美的驱动程序。

**作者：**驱动程序是由大的 Linux 公司研发还是由你个人编写，并不影响其是否能被内核接受。没有人对内核源码树享有特权。只要你充分了解内核社区，你就会发现这一点。

## 资源列表

**Linux 内核主源码树：** `ftp://kernel.org/pub/linux/kernel/…?? ==` 你的国家代码，例如“cn”、“us”、“uk”、“fr”等等

**Linux 内核邮件列表：** `linux-kernel@vger.kernel.org` [可通过向 `majordomo@vger.kernel.org` 发邮件来订阅]

**Linux 设备驱动程序，第三版（探讨 2.6.10 版内核）：** `https://lwn.net/Kernel/LDD3/` （免费版）

**LWN.net：** 每周内核开发活动摘要 - `https://lwn.net/`

2.6 版中 API 的变更：

<https://lwn.net/Articles/2.6-kernel-api/>

将旧版内核的驱动程序移植到 2.6 版：

<https://lwn.net/Articles/driver-porting/>

**内核新手 (KernelNewbies)**: 为新的内核开发者提供文档和帮助 <https://kernelnewbies.org/>

**Linux USB 项目**: <http://www.linux-usb.org/>

写内核驱动的“不要” (**Arjan van de Ven** 著)：<http://www.fenrus.org/how-to-not-write-a-device-driver.pdf>

**内核清洁工 (Kernel Janitor)**: <https://kernelnewbies.org/KernelJanitors>

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/process/submit-checklist.rst

**Translator** Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

## Linux 内核补丁提交清单

如果开发人员希望看到他们的内核补丁提交更快地被接受，那么他们应该做一些基本的事情。

这些都是在 *Documentation/translations/zh\_CN/process/submitting-patches.rst* 和其他有关提交 Linux 内核补丁的文档中提供的。

- 1) 如果使用工具，则包括定义/声明该工具的文件。不要依赖于其他头文件拉入您使用的头文件。
- 2) 干净的编译：
  - a) 使用适用或修改的 CONFIG 选项 =y、=m 和 =n。没有 GCC 警告/错误，没有链接器警告/错误。
  - b) 通过 allnoconfig、allmodconfig
  - c) 使用 0=builddir 时可以成功编译
- 3) 通过使用本地交叉编译工具或其他一些构建场在多个 CPU 体系结构上构建。
- 4) PPC64 是一种很好的交叉编译检查体系结构，因为它倾向于对 64 位的数使用无符号长整型。

- 5) 如下所述 [Documentation/translations/zh\\_CN/process/coding-style.rst](#)。检查您的补丁是否为常规样式。在提交 (`scripts/checkpatch.pl`) 之前，使用补丁样式检查器检查是否有轻微的冲突。您应该能够处理您的补丁中存在的所有违规行为。
- 6) 任何新的或修改过的 CONFIG 选项都不会弄脏配置菜单，并默认为关闭，除非它们符合 [Documentation/kbuild/kconfig-language.rst](#) 中记录的异常条件，菜单属性：默认值。
- 7) 所有新的 kconfig 选项都有帮助文本。
- 8) 已仔细审查了相关的 Kconfig 组合。这很难用测试来纠正——脑力在这里是有回报的。
- 9) 用 sparse 检查干净。
- 10) 使用 `make checkstack` 和 `make namespacecheck` 并修复他们发现的任何问题。

---

**Note:** `checkstack` 并没有明确指出问题，但是任何一个在堆栈上使用超过 512 字节的函数都可以进行更改。

---

- 11) 包括 kernel-doc 内核文档以记录全局内核 API。（静态函数不需要，但也可以。）使用 `make htmldocs` 或 `make pdfdocs` 检查 kernel-doc 并修复任何问题。
- 12) 通过以下选项同时启用的测试 `CONFIG_PREEMPT`, `CONFIG_DEBUG_PREEMPT`, `CONFIG_DEBUG_SLAB`, `CONFIG_DEBUG_PAGEALLOC`, `CONFIG_DEBUG_MUTEXES`, `CONFIG_DEBUG_SPINLOCK`, `CONFIG_DEBUG_ATOMIC_SLEEP`, `CONFIG_PROVE_RCU` 和 `CONFIG_DEBUG_OBJECTS_RCU_HEAD`
- 13) 已经过构建和运行时测试，包括有或没有 `CONFIG_SMP`, `CONFIG_PREEMPT`。
- 14) 如果补丁程序影响 IO/磁盘等：使用或不使用 `CONFIG_LBDAF` 进行测试。
- 15) 所有代码路径都已在启用所有 lockdep 功能的情况下运行。
- 16) 所有新的/proc 条目都记录在 [Documentation/](#)
- 17) 所有新的内核引导参数都记录在 [Documentation/admin-guide/kernel-parameters.rst](#) 中。
- 18) 所有新的模块参数都记录在 `MODULE_PARM_DESC()`
- 19) 所有新的用户空间接口都记录在 [Documentation/ABI/](#) 中。有关详细信息，请参阅 [Documentation/ABI/README](#)。更改用户空间接口的补丁应该抄送 [linux-api@vger.kernel.org](mailto:linux-api@vger.kernel.org)。
- 20) 已通过至少注入 slab 和 page 分配失败进行检查。请参阅 [Documentation/fault-injection/](#) 如果新代码是实质性的，那么添加子系统特定的故障注入可能是合适的。
- 21) 新添加的代码已经用 `gcc -W` 编译（使用 `make EXTRA_CFLAGS=-W`）。这将产生大量噪声，但对于查找诸如“警告：有符号和无符号之间的比较”之类的错误很有用。
- 22) 在它被合并到-mm 补丁集中之后进行测试，以确保它仍然与所有其他排队的补丁以及 VM、VFS 和其他子系统中的各种更改一起工作。
- 23) 所有内存屏障例如 `barrier()`, `rmb()`, `wmb()` 都需要源代码中的注释来解释它们正在执行的操作及其原因的逻辑。

- 24) 如果补丁添加了任何 ioctl，那么也要更新 Documentation/userspace-api/ioctl/ioctl-number.rst
- 25) 如果修改后的源代码依赖或使用与以下 Kconfig 符号相关的任何内核 API 或功能，则在禁用相关 Kconfig 符号和/或 =m（如果该选项可用）的情况下测试以下多个构建 [并非所有这些都同时存在，只是它们的各种/随机组合]：  
CONFIG\_SMP, CONFIG\_SYSFS, CONFIG\_PROC\_FS, CONFIG\_INPUT, CONFIG\_PCI, CONFIG\_BLOCK, CONFIG\_PM, CONFIG\_MAGIC\_SYSRQ, CONFIG\_NET, CONFIG\_INET=n (但是后者伴随 CONFIG\_NET=y).

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

### Original Documentation/process/stable-api-nonsense.rst

译者：

中文版维护者： 钟宇 TripleX Chung <xxx.phy@gmail.com>  
中文版翻译者： 钟宇 TripleX Chung <xxx.phy@gmail.com>  
中文版校译者： 李阳 Li Yang <leoyang.li@nxp.com>

## Linux 内核驱动接口

写作本文档的目的，是为了解释为什么 Linux 既没有二进制内核接口，也没有稳定的内核接口。这里所说的内核接口，是指内核里的接口，而不是内核和用户空间的接口。内核到用户空间的接口，是提供给应用程序使用的系统调用，系统调用在历史上几乎没有过变化，将来也不会有变化。我有一些老应用程序是在 0.9 版本或者更早版本的内核上编译的，在使用 2.6 版本内核的 Linux 发布上依然用得很好。用户和应用程序作者可以将这个接口看成是稳定的。

### 执行纲要

你也许以为自己想要稳定的内核接口，但是你不清楚你要的实际上不是它。你需要的其实是稳定的驱动程序，而你只有将驱动程序放到公版内核的源代码树里，才有可能达到这个目的。而且这样做还有很多其它好处，正是因为这些好处使得 Linux 能成为强壮，稳定，成熟的操作系统，这也是你最开始选择 Linux 的原因。

## 入门

只有那些写驱动程序的“怪人”才会担心内核接口的改变，对广大用户来说，既看不到内核接口，也不需要去关心它。

首先，我不打算讨论关于任何非 GPL 许可的内核驱动的法律问题，这些非 GPL 许可的驱动程序包括不公开源代码，隐藏源代码，二进制或者是用源代码包装，或者是其它任何形式的不能以 GPL 许可公开源代码的驱动程序。如果有法律问题，请咨询律师，我只是一个程序员，所以我只打算探讨技术问题（不是小看法律问题，法律问题很实际，并且需要一直关注）。

既然只谈技术问题，我们就有了下面两个主题：二进制内核接口和稳定的内核源代码接口。这两个问题是互相关联的，让我们先解决掉二进制接口的问题。

## 二进制内核接口

假如我们有一个稳定的内核源代码接口，那么自然而然的，我们就拥有了稳定的二进制接口，是这样的吗？错。让我们看看关于 Linux 内核的几点事实：

- 取决于所用的 C 编译器的版本，不同的内核数据结构里的结构体的对齐方式会有差别，代码中不同函数的表现形式也不一样（函数是不是被 `inline` 编译取决于编译器行为）。不同的函数的表现形式并不重要，但是数据结构内部的对齐方式很关键。
- 取决于内核的配置选项，不同的选项会让内核的很多东西发生改变：
  - 同一个结构体可能包含不同的成员变量
  - 有的函数可能根本不会被实现（比如编译的时候没有选择 SMP 支持一些锁函数就会被定义成空函数）。
  - 内核使用的内存会以不同的方式对齐，这取决于不同的内核配置选项。
- Linux 可以在很多的不同体系结构的处理器上运行。在某个体系结构上编译好的二进制驱动程序，不可能在另外一个体系结构上正确的运行。

对于一个特定的内核，满足这些条件并不难，使用同一个 C 编译器和同样的内核配置选项来编译驱动程序模块就可以了。这对于给一个特定 Linux 发布的特定版本提供驱动程序，是完全可以满足需求的。但是如果你要给不同发布的不同版本都发布一个驱动程序，就需要在每个发布上用不同的内核设置参数都编译一次内核，这简直跟噩梦一样。而且还要注意到，每个 Linux 发布还提供不同的 Linux 内核，这些内核都针对不同的硬件类型进行了优化（有很多种不同的处理器，还有不同的内核设置选项）。所以每发布一次驱动程序，都需要提供很多不同版本的内核模块。

相信我，如果你真的要采取这种发布方式，一定会慢慢疯掉，我很久以前就有过深刻的教训…

### 稳定的内核源代码接口

如果有人不将他的内核驱动程序，放入公版内核的源代码树，而又想让驱动程序一直保持在最新的内核中可用，那么这个话题将会变得没完没了。内核开发是持续而且快节奏的，从来都不会慢下来。内核开发人员在当前接口中找到 bug，或者找到更好的实现方式。一旦发现这些，他们就很快会去修改当前的接口。修改接口意味着，函数名可能会改变，结构体可能被扩充或者删减，函数的参数也可能发生改变。一旦接口被修改，内核中使用这些接口的地方需要同时修正，这样才能保证所有的东西继续工作。

举一个例子，内核的 USB 驱动程序接口在 USB 子系统的整个生命周期中，至少经历了三次重写。这些重写解决以下问题：

- 把数据流从同步模式改成非同步模式，这个改动减少了一些驱动程序的复杂度，提高了所有 USB 驱动程序的吞吐率，这样几乎所有的 USB 设备都能以最大速率工作了。
- 修改了 USB 核心代码中为 USB 驱动分配数据包内存的方式，所有的驱动都需要提供更多的参数给 USB 核心，以修正了很多已经被记录在案的死锁。

这和一些封闭源代码的操作系统形成鲜明的对比，在那些操作系统上，不得不额外的维护旧的 USB 接口。这导致了一个可能性，新的开发者依然会不小心使用旧的接口，以不恰当的方式编写代码，进而影响到操作系统的稳定性。在上面的例子中，所有的开发者都同意这些重要的改动，在这样的情况下修改代价很低。如果 Linux 保持一个稳定的内核源代码接口，那么就得创建一个新的接口；旧的，有问题的接口必须一直维护，给 Linux USB 开发者带来额外的工作。既然所有的 Linux USB 驱动的作者都是利用自己的时间工作，那么要求他们去做毫无意义的免费额外工作，是不可能的。安全问题对 Linux 来说十分重要。一个安全问题被发现，就会在短时间内得到修正。在很多情况下，这将导致 Linux 内核中的一些接口被重写，从根本上避免安全问题。一旦接口被重写，所有使用这些接口的驱动程序，必须同时得到修正，以确定安全问题已经得到修复并且不可能在未来还有同样的安全问题。如果内核内部接口不允许改变，那么就不可能修复这样的安全问题，也不可能确认这样的安全问题以后不会发生。开发者一直在清理内核接口。如果一个接口没有人在使用了，它就会被删除。这样可以确保内核尽可能的小，而且所有潜在的接口都会得到尽可能完整的测试（没有人使用的接口是不可能得到良好的测试的）。

### 要做什么

如果你写了一个 Linux 内核驱动，但是它还不在 Linux 源代码树里，作为一个开发者，你应该怎么做？为每个发布的每个版本提供一个二进制驱动，那简直是一个噩梦，要跟上永远处于变化之中的内核接口，也是一件辛苦活。很简单，让你的驱动进入内核源代码树（要记得我们在谈论的是以 GPL 许可发行的驱动，如果你的代码不符合 GPL，那么祝你好运，你只能自己解决这个问题了，你这个吸血鬼 < 把 Andrew 和 Linus 对吸血鬼的定义链接到这里 >）。当你的代码加入公版内核源代码树之后，如果一个内核接口改变，你的驱动会直接被修改接口的那个人修改。保证你的驱动永远都可以编译通过，并且一直工作，你几乎不需要做什么事情。

把驱动放到内核源代码树里会有很多的好处：

- 驱动的质量会提升，而维护成本（对原始作者来说）会下降。
- 其他人会给驱动添加新特性。
- 其他人会找到驱动中的 bug 并修复。

- 其他人会在驱动中找到性能优化的机会。
- 当外部的接口的改变需要修改驱动程序的时候，其他人会修改驱动程序
- 不需要联系任何发行商，这个驱动会自动的随着所有的 Linux 发布一起发布。

和别的操作系统相比，Linux 为更多不同的设备提供现成的驱动，而且能在更多不同体系结构的处理器上支持这些设备。这个经过考验的开发模式，必然是错不了的:)

## 感谢

感谢 Randy Dunlap, Andrew Morton, David Brownell, Hanna Linder, Robert Love, and Nishanth Aravamudan 对于本文档早期版本的评审和建议。

英文版维护者： Greg Kroah-Hartman <[greg@kroah.com](mailto:greg@kroah.com)>

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

## Original Documentation/process/stable-kernel-rules.rst

如果想评论或更新本文的内容，请直接联系原文档的维护者。如果你使用英文交流有困难的话，也可以向中文版维护者求助。如果本翻译更新不及时或者翻译存在问题，请联系中文版维护者：

中文版维护者： 钟宇 TripleX Chung <xxx.phy@gmail.com>  
中文版翻译者： 钟宇 TripleX Chung <xxx.phy@gmail.com>  
中文校版译者：  
- 李阳 Li Yang <leoyang.li@nxp.com>  
- Kangkai Yin <e12051@motorola.com>

## 所有你想知道的事情 - 关于 linux 稳定版发布

关于 Linux 2.6 稳定版发布，所有你想知道的事情。

### 关于哪些类型的补丁可以被接收进入稳定版代码树，哪些不可以的规则：

- 必须是显而易见的正确，并且经过测试的。
- 连同上下文，不能大于 100 行。
- 必须只修正一件事情。
- 必须修正了一个给大家带来麻烦的真正的 bug（不是“这也许是一个问题…”那样的东西）。
- 必须修正带来如下后果的问题：编译错误（对被标记为 CONFIG\_BROKEN 的例外），内核崩溃，挂起，数据损坏，真正的安全问题，或者一些类似“哦，这不好”的问题。简短的说，就是一些致命的问题。
- 没有“理论上的竞争条件”，除非能给出竞争条件如何被利用的解释。
- 不能存在任何的“琐碎的”修正（拼写修正，去掉多余空格之类的）。
- 必须被相关子系统的维护者接受。
- 必须遵循 Documentation/translations/zh\_CN/process/submitting-patches.rst 里的规则。

### 向稳定版代码树提交补丁的过程：

- 在确认了补丁符合以上的规则后，将补丁发送到 stable@vger.kernel.org。
- 如果补丁被接受到队列里，发送者会收到一个 ACK 回复，如果没有被接受，收到的是 NAK 回复。回复需要几天的时间，这取决于开发者的时间安排。
- 被接受的补丁会被加到稳定版本队列里，等待其他开发者的审查。
- 安全方面的补丁不要发到这个列表，应该发送到 security@kernel.org。

### 审查周期：

- 当稳定版的维护者决定开始一个审查周期，补丁将被发送到审查委员会，以及被补丁影响的领域的维护者（除非提交者就是该领域的维护者）并且抄送到 linux-kernel 邮件列表。
- 审查委员会有 48 小时的时间，用来决定给该补丁回复 ACK 还是 NAK。
- 如果委员会中有成员拒绝这个补丁，或者 linux-kernel 列表上有人反对这个补丁，并提出维护者和审查委员会之前没有意识到的问题，补丁会从队列中丢弃。
- 在审查周期结束的时候，那些得到 ACK 回应的补丁将会被加入到最新的稳定版发布中，一个新的稳定版发布就此产生。
- 安全性补丁将从内核安全小组那里直接接收到稳定版代码树中，而不是通过通常的审查周期。请联系内核安全小组以获得关于这个过程的更多细节。

## 审查委员会：

- 由一些自愿承担这项任务的内核开发者，和几个非志愿的组成。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/process/management-style.rst

**Translator** Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

## Linux 内核管理风格

这是一个简短的文档，描述了 Linux 内核首选的（或胡编的，取决于您问谁）管理风格。它的目的是在某种程度上参照 process/coding-style.rst 主要是为了避免反复回答<sup>1</sup> 相同（或类似）的问题。

管理风格是非常个人化的，比简单的编码风格规则更难以量化，因此本文档可能与实际情况有关，也可能与实际情况无关。起初它是一个玩笑，但这并不意味着它可能不是真的。你得自己决定。

顺便说一句，在谈到“核心管理者”时，主要是技术负责人，而不是在公司内部进行传统管理的人。如果你签署了采购订单或者对你的团队的预算有任何了解，你几乎肯定不是一个核心管理者。这些建议可能适用于您，也可能不适用于您。

首先，我建议你购买“高效人的七个习惯”，而不是阅读它。烧了它，这是一个伟大的象征性姿态。

不管怎样，这里是：

### 1) 决策

每个人都认为管理者做决定，而且决策很重要。决定越大越痛苦，管理者就必须越高级。这很明显，但事实并非如此。

最重要的是 避免做出决定。尤其是，如果有人告诉你“选择 (a) 或 (b)，我们真的需要你来做决定”，你就是陷入麻烦的管理者。你管理的人比你更了解细节，所以如果他们来找你做技术决策，你完蛋了。你显然没有能力为他们做这个决定。

（推论：如果你管理的人不比你更了解细节，你也会被搞砸，尽管原因完全不同。也就是说，你的工作是错的，他们应该管理你的才智）

<sup>1</sup> 本文件并不是通过回答问题，而是通过让提问者痛苦地明白，我们不知道答案是什么。

所以最重要的是 **避免**做出决定，至少是那些大而痛苦的决定。做一些小的和非结果性的决定是很好的，并且使您看起来好像知道自己在做什么，所以内核管理者需要做的是将那些大的和痛苦的决定变成那些没有人真正关心的小事情。

这有助于认识到一个大的决定和一个小的决定之间的关键区别是你是否可以在事后修正你的决定。任何决定都可以通过始终确保如果你错了（而且你一定会错），你以后总是可以通过回溯来弥补损失。突然间，你就要做两个无关紧要的决定，一个是错误的，另一个是正确的。

人们甚至会认为这是真正的领导能力（咳，胡说，咳）。

因此，避免重大决策的关键在于避免做那些无法挽回的事情。不要被引导到一个你无法逃离的角落。走投无路的老鼠可能很危险——走投无路的管理者真可怜。

事实证明，由于没有人会愚蠢到让内核管理者承担巨大的财政责任，所以通常很容易回溯。既然你不可能浪费掉你无法偿还的巨额资金，你唯一可以回溯的就是技术决策，而回溯很容易：只要告诉大家你是个不称职的傻瓜，说对不起，然后撤销你去年让别人所做的毫无价值的工作。突然间，你一年前做的决定不在是一个重大的决定，因为它很容易被推翻。

事实证明，有些人对接受这种方法有困难，原因有两个：

- 承认你是个白痴比看起来更难。我们都喜欢保持形象，在公共场合说你错了有时确实很难。
- 如果有人告诉你，你去年所做的工作终究是不值得的，那么对那些可怜的低级工程师来说也是很困难的，虽然实际的 **工作**很容易删除，但你可能已经不可挽回地失去了工程师的信任。记住：“**不可撤销**”是我们一开始就试图避免的，而你的决定终究是一个重大的决定。

令人欣慰的是，这两个原因都可以通过预先承认你没有任何线索，提前告诉人们你的决定完全是初步的，而且可能是错误的事情来有效地缓解。你应该始终保留改变主意的权利，并让人们 **意识到**这一点。当你 **还没有**做过真正愚蠢的事情的时候，承认自己是愚蠢的要容易得多。

然后，当它真的被证明是愚蠢的时候，人们就转动他们的眼珠说“哎呀，下次不要了”。

这种对不称职的先发制人的承认，也可能使真正做这项工作的人也会三思是否值得做。毕竟，如果他们不确定这是否是一个好主意，你肯定不应该通过向他们保证他们所做的工作将会进入（内核）鼓励他们。在他们开始一项巨大的努力之前，至少让他们三思而后行。

记住：他们最好比你更了解细节，而且他们通常认为他们对每件事都有答案。作为一个管理者，你能做的最好的事情不是灌输自信，而是对他们所做的事情进行健康的批判性思考。

顺便说一句，另一种避免做出决定的方法是看起来很可怜的抱怨“我们不能两者兼得吗？”相信我，它是有效的。如果不清楚哪种方法更好，他们最终会弄清楚的。最终的答案可能是两个团队都会因为这种情况而感到沮丧，以至于他们放弃了。

这听起来像是一个失败，但这通常是一个迹象，表明两个项目都有问题，而参与其中的人不能做决定的原因是他们都是错误的。你最终会闻到玫瑰的味道，你避免了另一个你本可以搞砸的决定。

## 2) 人

大多数人都很白痴，做一名管理者意味着你必须处理好这件事，也许更重要的是，**他们必须处理好你**。

事实证明，虽然很容易纠正技术错误，但不容易纠正人格障碍。你只能和他们的和你的（人格障碍）共处。

但是，为了做好作为内核管理者的准备，最好记住不要烧掉任何桥梁，不要轰炸任何无辜的村民，也不要疏远太多的内核开发人员。事实证明，疏远人是相当容易的，而亲近一个疏远的人是很难的。因此，“疏远”立即属于“不可逆”的范畴，并根据1) 决策 成为不可以做的事情。

这里只有几个简单的规则：

- (1) 不要叫人笨蛋（至少不要在公共场合）
- (2) 学习如何在忘记规则 (1) 时道歉

问题在于 #1 很容易去做，因为你可以用数百万种不同的方式说“你是一个笨蛋”<sup>2</sup> 有时甚至没有意识到，而且几乎总是带着一种白热化的信念，认为你是对的。

你越确信自己是对的（让我们面对现实吧，你可以把几乎所有人都称为坏人，而且你经常是对的），事后道歉就越难。

要解决此问题，您实际上只有两个选项：

- 非常擅长道歉
- 把“爱”均匀地散开，没有人会真正感觉到自己被不公平地瞄准了。让它有足够的创造性，他们甚至可能会觉得好笑。

选择永远保持礼貌是不存在的。没有人会相信一个如此明显地隐藏了他们真实性格的人。

## 3) 人 2 - 好人

虽然大多数人都很白痴，但不幸的是，据此推论你也是白痴，尽管我们都自我感觉良好，我们比普通人更好（让我们面对现实吧，没有人相信他们是普通人或低于普通人），我们也应该承认我们不是最锋利的刀，而且会有其他人比你更不像白痴。

有些人对聪明人反应不好。其他人利用它们。

作为内核维护人员，确保您在第二组中。接受他们，因为他们会让你的工作更容易。特别是，他们能够为你做决定，这就是游戏的全部内容。

所以当你发现一个比你聪明的人时，就顺其自然吧。你的管理职责在很大程度上变成了“听起来像是个好主意——去尝试吧”，或者“听起来不错，但是 XXX 呢？”“。第二个版本尤其是一个很好的方法，要么学习一些关于“XXX”的新东西，要么通过指出一些聪明人没有想到的东西来显得更具管理性。无论哪种情况，你都会赢。

要注意的一件事是认识到一个领域的伟大不一定会转化为其他领域。所以你可能会向特定的方向刺激人们，但让我们面对现实吧，他们可能擅长他们所做的事情，而且对其他事情都很差劲。好消息是，人们往往会自然而然地重拾他们擅长的东西，所以当你向某个方向刺激他们时，你并不是在做不可逆转的事情，只是不要用力推。

<sup>2</sup> 保罗·西蒙演唱了“离开爱人的 50 种方法”，因为坦率地说，“告诉开发者他们是 D\*CKHEAD”的 100 万种方法都无法确认。但我确信他已经这么想了。

### 4) 责备

事情会出问题的，人们希望去责备人。贴标签，你就是受责备的人。

事实上，接受责备并不难，尤其是当人们意识到这不全是你过错时。这让我们找到了承担责任的最佳方式：为别人承担这件事。你会感觉很好，他们会感觉很好，没有受到指责。那谁，失去了他们的全部36GB色情收藏的人，因为你的无能将勉强承认，你至少没有试图逃避责任。

然后让真正搞砸了的开发人员（如果你能找到他们）私下知道他们搞砸了。不仅是为了将来可以避免，而且为了让他们知道他们欠你一个人情。而且，也许更重要的是，他们也可能是能够解决问题的人。因为，让我们面对现实吧，肯定不是你。

承担责任也是你首先成为管理者的原因。这是让人们信任你，让你获得潜在的荣耀的一部分，因为你就是那个会说“我搞砸了”的人。如果你已经遵循了以前的规则，你现在已经很擅长说了。

### 5) 应避免的事情

有一件事人们甚至比被称为“笨蛋”更讨厌，那就是在一个神圣的声音中被称为“笨蛋”。第一个你可以道歉，第二个你不会真正得到机会。即使你做得很好，他们也可能不再倾听。

我们都认为自己比别人强，这意味着当别人装腔作势时，这会让我们很恼火。你也许在道德和智力上比你周围的每个人都优越，但不要试图太明显，除非你真的打算激怒某人<sup>3</sup>

同样，不要对事情太客气或太微妙。礼貌很容易落得落花流水，把问题隐藏起来，正如他们所说，“在互联网上，没人能听到你的含蓄。”用一个钝器把这一点锤进去，因为你不能真的依靠别人来获得你的观点。

一些幽默可以帮助缓和直率和道德化。过度到荒谬的地步，可以灌输一个观点，而不会让接受者感到痛苦，他们只是认为你是愚蠢的。因此，它可以帮助我们摆脱对批评的个人心理障碍。

### 6) 为什么是我？

既然你的主要责任似乎是为别人的错误承担责任，并且让别人痛苦地明白你是不称职的，那么显而易见的问题之一就变成了为什么首先要这样做。

首先，虽然你可能会或可能不会听到十几岁女孩（或男孩，让我们不要在这里评判或性别歧视）敲你的更衣室门，你会得到一个巨大的个人成就感为“负责”。别介意你真的在领导别人，你要跟上别人，尽可能快地追赶他们。每个人都会认为你是负责人。

如果你可以做到这个，这是个伟大的工作！

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

<sup>3</sup> 提示：与你的工作没有直接关系的网络新闻组是消除你对他人不满的好方法。偶尔写些侮辱性的帖子，打个喷嚏，让你的情绪得到净化。别把牢骚带回家

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题, 请联系该文件的译者, 或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/process/embargoed-hardware-issues.rst

**Translator** Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

## 被限制的硬件问题

### 范围

导致安全问题的硬件问题与只影响 Linux 内核的纯软件错误是不同的安全错误类别。

必须区别对待诸如熔毁 (Meltdown)、Spectre、L1TF 等硬件问题, 因为它们通常会影响所有操作系统 (“OS”), 因此需要在不同的 OS 供应商、发行版、硬件供应商和其他各方之间进行协调。对于某些问题, 软件缓解可能依赖于微码或固件更新, 这需要进一步的协调。

### 接触

Linux 内核硬件安全小组独立于普通的 Linux 内核安全小组。

该小组只负责协调被限制的硬件安全问题。Linux 内核中纯软件安全漏洞的报告不由该小组处理, 报告者将被引导至常规 Linux 内核安全小组 (Documentation/admin-guide/) 联系。

可以通过电子邮件 <[hardware-security@kernel.org](mailto:hardware-security@kernel.org)> 与小组联系。这是一份私密的安全官名单, 他们将帮助您根据我们的文档化流程协调问题。

邮件列表是加密的, 发送到列表的电子邮件可以通过 PGP 或 S/MIME 加密, 并且必须使用报告者的 PGP 密钥或 S/MIME 证书签名。该列表的 PGP 密钥和 S/MIME 证书可从 [https://www.kernel.org/…](https://www.kernel.org/) 获得。

虽然硬件安全问题通常由受影响的硬件供应商处理, 但我们欢迎发现潜在硬件缺陷的研究人员或个人与我们联系。

## 硬件安全官

目前的硬件安全官小组:

- Linus Torvalds (Linux 基金会院士)
- Greg Kroah Hartman (Linux 基金会院士)
- Thomas Gleixner (Linux 基金会院士)

### 邮件列表的操作

处理流程中使用的加密邮件列表托管在 Linux Foundation 的 IT 基础设施上。通过提供这项服务，Linux 基金会的 IT 基础设施安全总监在技术上有能力访问被限制的信息，但根据他的雇佣合同，他必须保密。Linux 基金会的 IT 基础设施安全总监还负责 kernel.org 基础设施。

Linux 基金会目前的 IT 基础设施安全总监是 Konstantin Ryabitsev。

### 保密协议

Linux 内核硬件安全小组不是正式的机构，因此无法签订任何保密协议。核心社区意识到这些问题的敏感性，并提供了一份谅解备忘录。

### 谅解备忘录

Linux 内核社区深刻理解在不同操作系统供应商、发行商、硬件供应商和其他各方之间进行协调时，保持硬件安全问题处于限制状态的要求。

Linux 内核社区在过去已经成功地处理了硬件安全问题，并且有必要的机制允许在限制限制下进行符合社区的开发。

Linux 内核社区有一个专门的硬件安全小组负责初始联系，并监督在限制规则下处理此类问题的过程。

硬件安全小组确定开发人员（领域专家），他们将组成特定问题的初始响应小组。最初的响应小组可以引入更多的开发人员（领域专家）以最佳的技术方式解决这个问题。

所有相关开发商承诺遵守限制规定，并对收到的信息保密。违反承诺将导致立即从当前问题中排除，并从所有相关邮件列表中删除。此外，硬件安全小组还将把违反者排除在未来的问题之外。这一后果的影响在我们社区是一种非常有效的威慑。如果发生违规情况，硬件安全小组将立即通知相关方。如果您或任何人发现潜在的违规行为，请立即向硬件安全人员报告。

### 流程

由于 Linux 内核开发的全球分布式特性，面对面的会议几乎不可能解决硬件安全问题。由于时区和其他因素，电话会议很难协调，只能在绝对必要时使用。加密电子邮件已被证明是解决此类问题的最有效和最安全的通信方法。

## 开始披露

披露内容首先通过电子邮件联系 Linux 内核硬件安全小组。此初始联系人应包含问题的描述和任何已知受影响硬件的列表。如果您的组织制造或分发受影响的硬件，我们建议您也考虑哪些其他硬件可能会受到影响。

硬件安全小组将提供一个特定于事件的加密邮件列表，用于与报告者进行初步讨论、进一步披露和协调。

硬件安全小组将向披露方提供一份开发人员（领域专家）名单，在与开发人员确认他们将遵守本谅解备忘录和文件化流程后，应首先告知开发人员有关该问题的信息。这些开发人员组成初始响应小组，并在初始接触后负责处理问题。硬件安全小组支持响应小组，但不一定参与缓解开发过程。

虽然个别开发人员可能通过其雇主受到保密协议的保护，但他们不能以 Linux 内核开发人员的身份签订个别保密协议。但是，他们将同意遵守这一书面程序和谅解备忘录。

披露方应提供已经或应该被告知该问题的所有其他实体的联系人名单。这有几个目的：

- 披露的实体列表允许跨行业通信，例如其他操作系统供应商、硬件供应商等。
- 可联系已披露的实体，指定应参与缓解措施开发的专家。
- 如果需要处理某一问题的专家受雇于某一上市实体或某一上市实体的成员，则响应小组可要求该实体披露该专家。这确保专家也是实体反应小组的一部分。

## 披露

披露方通过特定的加密邮件列表向初始响应小组提供详细信息。

根据我们的经验，这些问题的技术文档通常是一个足够的起点，最好通过电子邮件进行进一步的技术澄清。

## 缓解开发

初始响应小组设置加密邮件列表，或在适当的情况下重新修改现有邮件列表。

使用邮件列表接近于正常的 Linux 开发过程，并且在过去已经成功地用于为各种硬件安全问题开发缓解措施。

邮件列表的操作方式与正常的 Linux 开发相同。发布、讨论和审查修补程序，如果同意，则应用于非公共 git 存储库，参与开发人员只能通过安全连接访问该存储库。存储库包含针对主线内核的主开发分支，并根据需要为稳定的内核版本提供向后移植分支。

最初的响应小组将根据需要从 Linux 内核开发人员社区中确定更多的专家。引进专家可以在开发过程中的任何时候发生，需要及时处理。

如果专家受雇于披露方提供的披露清单上的实体或其成员，则相关实体将要求其参与。

否则，披露方将被告知专家参与的情况。谅解备忘录涵盖了专家，要求披露方确认参与。如果披露方有令人信服的理由提出异议，则必须在五个个工作日内提出异议，并立即与事件小组解决。如果披露方在五个个工作日内未作出回应，则视为默许。

在确认或解决异议后，专家由事件小组披露，并进入开发过程。

### 协调发布

有关各方将协商限制结束的日期和时间。此时，准备好的缓解措施集成到相关的内核树中并发布。

虽然我们理解硬件安全问题需要协调限制时间，但限制时间应限制在所有有关各方制定、测试和准备缓解措施所需的最短时间内。人为地延长限制时间以满足会议讨论日期或其他非技术原因，会给相关的开发人员和响应小组带来了更多的工作和负担，因为补丁需要保持最新，以便跟踪正在进行的上游内核开发，这可能会造成冲突的更改。

### CVE 分配

硬件安全小组和初始响应小组都不分配 CVE，开发过程也不需要 CVE。如果 CVE 是由披露方提供的，则可用于文档中。

### 流程专使

为了协助这一进程，我们在各组织设立了专使，他们可以回答有关报告流程和进一步处理的问题或提供指导。专使不参与特定问题的披露，除非响应小组或相关披露方提出要求。现任专使名单：

ARM	
AMD	Tom Lendacky < <a href="mailto:tom.lendacky@amd.com">tom.lendacky@amd.com</a> >
IBM	
Intel	Tony Luck < <a href="mailto:tony.luck@intel.com">tony.luck@intel.com</a> >
Qualcomm	Trilok Soni < <a href="mailto:tsoni@codeaurora.org">tsoni@codeaurora.org</a> >
Microsoft	Sasha Levin < <a href="mailto:sashal@kernel.org">sashal@kernel.org</a> >
VMware	
Xen	Andrew Cooper < <a href="mailto:andrew.cooper3@citrix.com">andrew.cooper3@citrix.com</a> >
Canonical	John Johansen < <a href="mailto:john.johansen@canonical.com">john.johansen@canonical.com</a> >
Debian	Ben Hutchings < <a href="mailto:ben@decadent.org.uk">ben@decadent.org.uk</a> >
Oracle	Konrad Rzeszutek Wilk < <a href="mailto:konrad.wilk@oracle.com">konrad.wilk@oracle.com</a> >
Red Hat	Josh Poimboeuf < <a href="mailto:jpoimboe@redhat.com">jpoimboe@redhat.com</a> >
SUSE	Jiri Kosina < <a href="mailto:jkosina@suse.cz">jkosina@suse.cz</a> >
Amazon	
Google	Kees Cook < <a href="mailto:keescook@chromium.org">keescook@chromium.org</a> >

如果要将您的组织添加到专使名单中，请与硬件安全小组联系。被提名的专使必须完全理解和支持我们的过程，并且在 Linux 内核社区中很容易联系。

## 加密邮件列表

我们使用加密邮件列表进行通信。这些列表的工作原理是，发送到列表的电子邮件使用列表的 PGP 密钥或列表的/MIME 证书进行加密。邮件列表软件对电子邮件进行解密，并使用订阅者的 PGP 密钥或 S/MIME 证书为每个订阅者分别对其进行重新加密。有关邮件列表软件和用于确保列表安全和数据保护的设置的详细信息，请访问: <https://www.kernel.org/>…

## 关键点

初次接触见[接触](#)。对于特定于事件的邮件列表，密钥和 S/MIME 证书通过特定列表发送的电子邮件传递给订阅者。

## 订阅事件特定列表

订阅由响应小组处理。希望参与通信的披露方将潜在订户的列表发送给响应组，以便响应组可以验证订阅请求。每个订户都需要通过电子邮件向响应小组发送订阅请求。电子邮件必须使用订阅服务器的 PGP 密钥或 S/MIME 证书签名。如果使用 PGP 密钥，则必须从公钥服务器获得该密钥，并且理想情况下该密钥连接到 Linux 内核的 PGP 信任网。另请参见: <https://www.kernel.org/signature.html>.

响应小组验证订阅者，并将订阅者添加到列表中。订阅后，订阅者将收到来自邮件列表的电子邮件，该邮件列表使用列表的 PGP 密钥或列表的/MIME 证书签名。订阅者的电子邮件客户端可以从签名中提取 PGP 密钥或 S/MIME 证书，以便订阅者可以向列表发送加密电子邮件。

这些是一些总体性技术指南，由于不大好分类而放在这里：

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

### Original Documentation/process/magic-number.rst

如果想评论或更新本文的内容，请直接发信到 LKML。如果你使用英文交流有困难的话，也可以向中文版维护者求助。如果本翻译更新不及时或者翻译存在问题，请联系中文版维护者：

中文版维护者： 贾威威 Jia Wei Wei < <a href="mailto:harryxiyou@gmail.com">harryxiyou@gmail.com</a> >
中文版翻译者： 贾威威 Jia Wei Wei < <a href="mailto:harryxiyou@gmail.com">harryxiyou@gmail.com</a> >
中文版校译者： 贾威威 Jia Wei Wei < <a href="mailto:harryxiyou@gmail.com">harryxiyou@gmail.com</a> >

### Linux 魔术数

这个文件是有关当前使用的魔术值注册表。当你给一个结构添加了一个魔术值，你也应该把这个魔术值添加到这个文件，因为我们最好把用于各种结构的魔术值统一起来。

使用魔术值来保护内核数据结构是一个非常好的主意。这就允许你在运行期检查 (a) 一个结构是否已经被攻击，或者 (b) 你已经给一个例行程序通过了一个错误的结构。后一种情况特别地有用—特别是当你通过一个空指针指向结构体的时候。tty 源码，例如，经常通过特定驱动使用这种方法并且反复地排列特定方面的结构。

使用魔术值的方法是在结构的开始处声明的，如下：

```
struct tty_ldisc {  
    int     magic;  
    ...  
};
```

当你以后给内核添加增强功能的时候，请遵守这条规则！这样就会节省数不清的调试时间，特别是一些古怪的情况，例如，数组超出范围并且重新写了超出部分。遵守这个规则，这些情况可以被快速地，安全地避免。

**Theodore Ts' o** 31 Mar 94

给当前的 Linux 2.1.55 添加魔术表。

Michael Chastain <mailto:mec@shout.net> 22 Sep 1997

现在应该最新的 Linux 2.1.112。因为在特性冻结期间，不能在 2.2.x 前改变任何东西。这些条目被数域所排序。

Krzysztof G.Baranowski <mailto: kgb@knm.org.pl> 29 Jul 1998

更新魔术表到 Linux 2.5.45。刚好越过特性冻结，但是有可能还会有一些新的魔术值在 2.6.x 之前融入到内核中。

Petr Baudis <mailto:pasky@ucw.cz> 03 Nov 2002

更新魔术表到 Linux 2.5.74。

Fabian Frederick <mailto:ffrederick@users.sourceforge.net> 09 Jul 2003

魔术数名	数字	结构	文件
PG_MAGIC	'P'	pg_{read,write}_hdr	include/linux/pg.h
CMAGIC	0x0111	user	include/linux/a.out.
MKISS_DRIVER_MAGIC	0x04bf	mkiss_channel	drivers/net/mkiss.h
HDLC_MAGIC	0x239e	n_hdlc	drivers/char/n_hdlc.
APM_BIOS_MAGIC	0x4101	apm_user	arch/x86/kernel/apm_
DB_MAGIC	0x4442	fc_info	drivers/net/iph5526_
DL_MAGIC	0x444d	fc_info	drivers/net/iph5526_
FASYNC_MAGIC	0x4601	fasync_struct	include/linux/fs.h
FF_MAGIC	0x4646	fc_info	drivers/net/iph5526_

Table 2 – continued from previous page

魔术数名	数字	结构	文件
PTY_MAGIC	0x5001		drivers/char/pty.c
PPP_MAGIC	0x5002	ppp	include/linux/if_ppp.h
SSTATE_MAGIC	0x5302	serial_state	include/linux/serial.h
SLIP_MAGIC	0x5302	slip	drivers/net/sliph.h
STRIP_MAGIC	0x5303	strip	drivers/net/strip.c
SIXPACK_MAGIC	0x5304	sixpack	drivers/net/hamradio.h
AX25_MAGIC	0x5316	ax_disp	drivers/net/mkiss.h
TTY_MAGIC	0x5401	tty_struct	include/linux/tty.h
MGSL_MAGIC	0x5401	mgsl_info	drivers/char/synclink.h
TTY_DRIVER_MAGIC	0x5402	tty_driver	include/linux/tty_driver.h
MGSLPC_MAGIC	0x5402	mgslpc_info	drivers/char/pcmcia.h
USB_SERIAL_MAGIC	0x6702	usb_serial	drivers/usb/serial/usb.h
FULL_DUPLEX_MAGIC	0x6969		drivers/net/ethernet.h
USB_BLUETOOTH_MAGIC	0x6d02	usb_bluetooth	drivers/usb/class/bluetooth.h
RFCOMM_TTY_MAGIC	0x6d02		net/bluetooth/rfcomm.h
USB_SERIAL_PORT_MAGIC	0x7301	usb_serial_port	drivers/usb/serial/usb.h
CG_MAGIC	0x00090255	ufs_cylinder_group	include/linux/ufs_fs.h
LSEMAGIC	0x05091998	lse	drivers/fc4/fc.c
GDTIOCTL_MAGIC	0x06030f07	gdth_iowr_str	drivers/scsi/gdth_ioctl.h
RIEBL_MAGIC	0x09051990		drivers/net/atari_lan.h
NBD_REQUEST_MAGIC	0x12560953	nbd_request	include/linux/nbd.h
RED_MAGIC2	0x170fc2a5	(any)	mm/slab.c
BAYCOM_MAGIC	0x19730510	baycom_state	drivers/net/baycom_e.h
ISDN_X25IFACE_MAGIC	0x1e75a2b9	isdn_x25iface_proto_data	drivers/isdn/isdn_x25iface.h
ECP_MAGIC	0x21504345	cdkecpsig	include/linux/cdk.h
LSOMAGIC	0x27091997	lso	drivers/fc4/fc.c
LSMAGIC	0x2a3b4d2a	ls	drivers/fc4/fc.c
WANPIPE_MAGIC	0x414C4453	sdla_{dump,exec}	include/linux/wanpipe.h
CS_CARD_MAGIC	0x43525553	cs_card	sound/oss/cs46xx.c
LABELCL_MAGIC	0x4857434c	labelcl_info_s	include/asm/ia64/sn.h
ISDN_ASYNC_MAGIC	0x49344C01	modem_info	include/linux/isdn.h
CTC_ASYNC_MAGIC	0x49344C01	ctc_tty_info	drivers/s390/net/ctc.h
ISDN_NET_MAGIC	0x49344C02	isdn_net_local_s	drivers/isdn/i4l/isdn.h
SAVEKMSG_MAGIC2	0x4B4D5347	savekmsg	arch/*/amiga/config.h
CS_STATE_MAGIC	0x4c4f4749	cs_state	sound/oss/cs46xx.c
SLAB_C_MAGIC	0x4f17a36d	kmem_cache	mm/slab.c
COW_MAGIC	0x4f4f4f4d	cow_header_v1	arch/um/drivers/ubd.h
I810_CARD_MAGIC	0x5072696E	i810_card	sound/oss/i810_audio.h

con

Table 2 – continued from previous page

魔术数名	数字	结构	文件
TRIDENT_CARD_MAGIC	0x5072696E	trident_card	sound/oss/trident.c
ROUTER_MAGIC	0x524d4157	wan_device	[in wanrouter.h pre 3]
SAVEKMSG_MAGIC1	0x53415645	savekmsg	arch/*/amiga/config.
GDA_MAGIC	0x58464552	gda	arch/mips/include/as
RED_MAGIC1	0x5a2cf071	(any)	mm/slab.c
EEPROM_MAGIC_VALUE	0x5ab478d2	lanai_dev	drivers/atm/lanai.c
HDLCDRV_MAGIC	0x5ac6e778	hdlcdrv_state	include/linux/hlcdrv
PCXX_MAGIC	0x5c6df104	channel	drivers/char/pcxx.h
KV_MAGIC	0x5f4b565f	kernel_vars_s	arch/mips/include/as
I810_STATE_MAGIC	0x63657373	i810_state	sound/oss/i810_audio
TRIDENT_STATE_MAGIC	0x63657373	trient_state	sound/oss/trident.c
M3_CARD_MAGIC	0x646e6f50	m3_card	sound/oss/maestro3.c
FW_HEADER_MAGIC	0x65726F66	fw_header	drivers/atm/fore200e
SLOT_MAGIC	0x67267321	slot	drivers/hotplug/cpqpp
SLOT_MAGIC	0x67267322	slot	drivers/hotplug/acpi
LO_MAGIC	0x68797548	nbd_device	include/linux/nbd.h
M3_STATE_MAGIC	0x734d724d	m3_state	sound/oss/maestro3.c
VMALLOC_MAGIC	0x87654320	snd_alloc_track	sound/core/memory.c
KMALLOC_MAGIC	0x87654321	snd_alloc_track	sound/core/memory.c
PWC_MAGIC	0x89DC10AB	pwc_device	drivers/usb/media/pw
NBD_REPLY_MAGIC	0x96744668	nbd_reply	include/linux/nbd.h
ENI155_MAGIC	0xa54b872d	midway_eprom	drivers/atm/eni.h
CODA_MAGIC	0xC0DAC0DA	coda_file_info	fs/coda/coda_fs_i.h
DPMEM_MAGIC	0xc0ffee11	gdt_pci_sram	drivers/scsi/gdth.h
YAM_MAGIC	0xF10A7654	yam_port	drivers/net/hamradio
CCB_MAGIC	0xf2691ad2	ccb	drivers/scsi/ncr53c8
QUEUE_MAGIC_FREE	0xf7e1c9a3	queue_entry	drivers/scsi/arm/que
QUEUE_MAGIC_USED	0xf7e1cc33	queue_entry	drivers/scsi/arm/que
HTB_CMAGIC	0xFEFAFEF1	htb_class	net/sched/sch_htb.c
NMI_MAGIC	0x48414d4d455201	nmi_s	arch/mips/include/as

请注意，在声音记忆管理中仍然有一些特殊的为每个驱动定义的魔术值。查看 include/sound/sndmagic.h 来获取他们完整的列表信息。很多 OSS 声音驱动拥有自己从声卡 PCI ID 构建的魔术值-他们也没有被列在这里。

IrDA 子系统也使用了大量的自己的魔术值，查看 include/net/irda/irda.h 来获取他们完整的信息。

HFS 是另外一个比较大的使用魔术值的文件系统-你可以在 fs/hfs/hfs.h 中找到他们。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

### Original Documentation/process/volatile-considered-harmful.rst

如果想评论或更新本文的内容，请直接联系原文档的维护者。如果你使用英文交流有困难的话，也可以向中文版维护者求助。如果本翻译更新不及时或者翻译存在问题，请联系中文版维护者：

英文版维护者： Jonathan Corbet <[corbet@lwn.net](mailto:corbet@lwn.net)>  
 中文版维护者： 伍鹏 Bryan Wu <[bryan.wu@analog.com](mailto:bryan.wu@analog.com)>  
 中文版翻译者： 伍鹏 Bryan Wu <[bryan.wu@analog.com](mailto:bryan.wu@analog.com)>  
 中文版校译者： 张汉辉 Eugene Teo <[eugeneteo@kernel.sg](mailto:eugeneteo@kernel.sg)>  
     杨瑞 Dave Young <[hidave.darkstar@gmail.com](mailto:hidave.darkstar@gmail.com)>  
     时奎亮 Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

## 为什么不应该使用“volatile”类型

C 程序员通常认为 volatile 表示某个变量可以在当前执行的线程之外被改变；因此，在内核中用到共享数据结构时，常常会有 C 程序员喜欢使用 volatile 这类变量。换句话说，他们经常会把 volatile 类型看成某种简易的原子变量，当然它们不是。在内核中使用 volatile 几乎总是错误的；本文档将解释为什么这样。

理解 volatile 的关键是知道它的目的是用来消除优化，实际上很少有人真正需要这样的应用。在内核中，程序员必须防止意外的并发访问破坏共享的数据结构，这其实是一个完全不同的任务。用来防止意外并发访问的保护措施，可以更加高效的避免大多数优化相关的问题。

像 volatile 一样，内核提供了很多原语来保证并发访问时的数据安全（自旋锁，互斥量，内存屏障等等），同样可以防止意外的优化。如果可以正确使用这些内核原语，那么就没有必要再使用 volatile。如果仍然必须使用 volatile，那么几乎可以肯定在代码的某处有一个 bug。在正确设计的内核代码中，volatile 能带来的仅仅是使事情变慢。

思考一下这段典型的内核代码：

```
spin_lock(&the_lock);
do_something_on(&shared_data);
do_something_else_with(&shared_data);
spin_unlock(&the_lock);
```

如果所有的代码都遵循加锁规则，当持有 the\_lock 的时候，不可能意外的改变 shared\_data 的值。任何可能访问该数据的其他代码都会在这个锁上等待。自旋锁原语跟内存屏障一样——它们显式的用来书写成这样——意味着数据访问不会跨越它们而被优化。所以本来编译器认为它知道在 shared\_data 里面将有什么，但

是因为 `spin_lock()` 调用跟内存屏障一样，会强制编译器忘记它所知道的一切。那么在访问这些数据时不会有优化的问题。

如果 `shared_data` 被声明为 `volatile`，锁操作将仍然是必须的。就算我们知道没有其他人正在使用它，编译器也将被阻止优化对临界区内 `shared_data` 的访问。在锁有效的同时，`shared_data` 不是 `volatile` 的。在处理共享数据的时候，适当的锁操作可以不再需要 `volatile` ——并且是有潜在危害的。

`volatile` 的存储类型最初是为那些内存映射的 I/O 寄存器而定义。在内核里，寄存器访问也应该被锁保护，但是人们也不希望编译器“优化”临界区内的寄存器访问。内核里 I/O 的内存访问是通过访问函数完成的；不赞成通过指针对 I/O 内存的直接访问，并且不是在所有体系架构上都能工作。那些访问函数正是为了防止意外优化而写的，因此，再说一次，`volatile` 类型不是必需的。

另一种引起用户可能使用 `volatile` 的情况是当处理器正忙着等待一个变量的值。正确执行一个忙等待的方法是：

```
while (my_variable != what_i_want)
    cpu_relax();
```

`cpu_relax()` 调用会降低 CPU 的能量消耗或者让位于超线程双处理器；它也作为内存屏障一样出现，所以，再一次，`volatile` 不是必需的。当然，忙等待一开始就是一种反常规的做法。

在内核中，一些稀少的情况下 `volatile` 仍然是有意义的：

- 在一些体系架构的系统上，允许直接的 I/O 内存访问，那么前面提到的访问函数可以使用 `volatile`。基本上，每一个访问函数调用它自己都是一个小的临界区域并且保证了按照程序员期望的那样发生访问操作。
- 某些会改变内存的内联汇编代码虽然没有什么其他明显的附作用，但是有被 GCC 删除的可能性。在汇编声明中加上 `volatile` 关键字可以防止这种删除操作。
- Jiffies 变量是一种特殊情况，虽然每次引用它的时候都可以有不同的值，但读 `jiffies` 变量时不需要任何特殊的加锁保护。所以 `jiffies` 变量可以使用 `volatile`，但是不赞成其他跟 `jiffies` 相同类型变量使用 `volatile`。Jiffies 被认为是一种“愚蠢的遗留物”（Linus 的话）因为解决这个问题比保持现状要麻烦的多。
- 由于某些 I/O 设备可能会修改连续一致的内存，所以有时，指向连续一致内存的数据结构的指针需要正确的使用 `volatile`。网络适配器使用的环状缓存区正是这类情形的一个例子，其中适配器用改变指针来表示哪些描述符已经处理过了。

对于大多代码，上述几种可以使用 `volatile` 的情况都不适用。所以，使用 `volatile` 是一种 bug 并且需要对这样的代码额外仔细检查。那些试图使用 `volatile` 的开发人员需要退一步想想他们真正想实现的是什么。

非常欢迎删除 `volatile` 变量的补丁——只要证明这些补丁完整的考虑了并发问题。

## 注释

[1] <https://lwn.net/Articles/233481/> [2] <https://lwn.net/Articles/233482/>

## 致谢

最初由 Randy Dunlap 推动并作初步研究由 Jonathan Corbet 撰写参考 Satyam Sharma, Johannes Stezenbach, Jesper Juhl, Heikki Orsila, H. Peter Anvin, Philipp Hahn 和 Stefan Richter 的意见改善了本档。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/dev-tools/index.rst

**Translator** 赵军奎 Bernard Zhao <[bernard@vivo.com](mailto:bernard@vivo.com)>

## \* 内核开发工具

本文档是有关内核开发工具文档的合集。目前这些文档已经整理在一起，不需要再花费额外的精力。欢迎任何补丁。

有关测试专用工具的简要概述，参见[内核测试指南](#)

目录

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/dev-tools/testing-overview.rst

**Translator** 胡皓文 Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

### 内核测试指南

有许多不同的工具可以用于测试 Linux 内核，因此了解什么时候使用它们可能很困难。本文档粗略概述了它们之间的区别，并阐释了它们是怎样糅合在一起的。

### 编写和运行测试

大多数内核测试都是用 `kselbstest` 或 KUnit 框架之一编写的。它们都让运行测试更加简化，并为编写新测试提供帮助。

如果你想验证内核的行为——尤其是内核的特定部分——那你就要使用 KUnit 或 `kselbstest`。

### KUnit 和 `kselbstest` 的区别

---

**Note:** 由于本文段中部分术语尚无较好的对应中文释义，可能导致与原文含义存在些许差异，因此建议读者结合原文（Documentation/dev-tools/testing-overview.rst）辅助阅读。如对部分翻译有异议或有更好的翻译意见，欢迎联系译者进行修订。

---

KUnit（Documentation/dev-tools/kunit/index.rst）是用于“白箱”测试的一个完整的内核内部系统：因为测试代码是内核的一部分，所以它能够访问用户空间不能访问到的内部结构和功能。

因此，KUnit 测试最好针对内核中较小的、自包含的部分，以便能够独立地测试。“单元”测试的概念亦是如此。

比如，一个 KUnit 测试可能测试一个单独的内核功能（甚至通过一个函数测试一个单一的代码路径，例如一个错误处理案例），而不是整个地测试一个特性。

这也使得 KUnit 测试构建和运行非常地快，从而能够作为开发流程的一部分被频繁地运行。

有关更详细的介绍，请参阅 KUnit 测试代码风格指南 Documentation/dev-tools/kunit/style.rst

`kselbstest`（Documentation/dev-tools/kselbstest.rst），相对来说，大量用于用户空间，并且通常测试用户空间的脚本或程序。

这使得编写复杂的测试，或者需要操作更多全局系统状态的测试更加容易（诸如生成进程之类）。然而，从 `kselbstest` 直接调用内核函数是不行的。这也就意味着只有通过某种方式（如系统调用、驱动设备、文件系统等）导出到了用户空间的内核功能才能使用 `kselbstest` 来测试。为此，有些测试包含了一个伴生的内核模块用于导出更多的信息和功能。不过，对于基本上或者完全在内核中运行的测试，KUnit 可能是更佳工具。

`kselbstest` 也因此非常适合于全部功能的测试，因为这些功能会将接口暴露到用户空间，从而能够被测试，而不是展现实现细节。“system” 测试和 “end-to-end” 测试亦是如此。

比如，一个新的系统调用应该伴随有新的 `kselbstest` 测试。

## 代码覆盖率工具

支持两种不同代码之间的覆盖率测量工具。它们可以用来验证一项测试执行的确切函数或代码行。这有助于决定内核被测试了多少，或用来查找合适的测试中没有覆盖到的极端情况。

[在 Linux 内核里使用 gcov 做代码覆盖率检查](#) 是 GCC 的覆盖率测试工具，能用于获取内核的全局或每个模块的覆盖率。与 KCOV 不同的是，这个工具不记录每个任务的覆盖率。覆盖率数据可以通过 debugfs 读取，并通过常规的 gcov 工具进行解释。

Documentation/dev-tools/kcov.rst 是能够构建在内核之中，用于在每个任务的层面捕捉覆盖率的一个功能。因此，它对于模糊测试和关于代码执行期间信息的其它情况非常有用，比如在一个单一系统调用里使用它就很有用。

## 动态分析工具

内核也支持许多动态分析工具，用以检测正在运行的内核中出现的多种类型的问题。这些工具通常每个去寻找一类不同的缺陷，比如非法内存访问，数据竞争等并发问题，或整型溢出等其他未定义行为。

如下所示：

- kmemleak 检测可能的内存泄漏。参阅 Documentation/dev-tools/kmemleak.rst
- KASAN 检测非法内存访问，如数组越界和释放后重用（UAF）。参阅 Documentation/dev-tools/kasan.rst
- UBSAN 检测 C 标准中未定义的行为，如整型溢出。参阅 Documentation/dev-tools/ubsan.rst
- KCSAN 检测数据竞争。参阅 Documentation/dev-tools/kcsan.rst
- KFENCE 是一个低开销的内存问题检测器，比 KASAN 更快且能被用于批量构建。参阅 Documentation/dev-tools/kfence.rst
- lockdep 是一个锁定正确性检测器。参阅 Documentation/locking/lockdep-design.rst
- 除此以外，在内核中还有一些其它的调试工具，大多数能在 lib/Kconfig.debug 中找到。

这些工具倾向于对内核进行整体测试，并且不像 kselftest 和 KUnit 一样“传递”。它们可以通过在启用这些工具时运行内核测试以与 kselftest 或 KUnit 结合起来：之后你就能确保这些错误在测试过程中都不会发生了。

一些工具与 KUnit 和 kselftest 集成，并且在检测到问题时会自动打断测试。

Copyright 2004 Linus Torvalds Copyright 2004 Pavel Macheck <[pavel@ucw.cz](mailto:pavel@ucw.cz)> Copyright 2006 Bob Copeland <[me@bobcopeland.com](mailto:me@bobcopeland.com)>

## orphan

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题, 请联系该文件的译者, 或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

## Original Documentation/dev-tools/sparse.rst

翻译 Li Yang <[leoyang.li@nxp.com](mailto:leoyang.li@nxp.com)>

校译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## Sparse

Sparse 是一个 C 程序的语义检查器; 它可以用来发现内核代码的一些潜在问题。关于 sparse 的概述, 请参见 <https://lwn.net/Articles/689907/>; 本文档包含一些针对内核的 sparse 信息。关于 sparse 的更多信息, 主要是关于它的内部结构, 可以在它的官方网页上找到: <https://sparse.docs.kernel.org>。

## 使用 sparse 工具做类型检查

“`_bitwise`”是一种类型属性, 所以你应该这样使用它:

```
typedef int __bitwise pm_request_t;

enum pm_request {
    PM_SUSPEND = (__force pm_request_t) 1,
    PM_RESUME = (__force pm_request_t) 2
};
```

这样会使 `PM_SUSPEND` 和 `PM_RESUME` 成为位方式 (`bitwise`) 整数 (使用”`_force`”是因为 `sparse` 会抱怨改变位方式的类型转换, 但是这里我们确实需要强制进行转换)。而且因为所有枚举值都使用了相同的类型, 这里的”`enum pm_request`”也将会使用那个类型做为底层实现。

而且使用 `gcc` 编译的时候, 所有的 `_bitwise/_force` 都会消失, 最后在 `gcc` 看来它们只不过是普通的整数。

坦白来说, 你并不需要使用枚举类型。上面那些实际都可以浓缩成一个特殊的”`int _bitwise`”类型。

所以更简单的办法只要这样做:

```
typedef int __bitwise pm_request_t;

#define PM_SUSPEND ((__force pm_request_t) 1)
#define PM_RESUME ((__force pm_request_t) 2)
```

现在你就有了严格的类型检查所需要的所有基础架构。

一个小提醒: 常数整数”0”是特殊的。你可以直接把常数零当作位方式整数使用而不用担心 `sparse` 会抱怨。这是因为”`bitwise`”(恰如其名)是用来确保不同位方式类型不会被弄混 (小尾模式, 大尾模式, `cpu` 尾模式, 或者其他), 对他们来说常数”0”确实是特殊的。

## 使用 sparse 进行锁检查

下面的宏对于 gcc 来说是未定义的，在 sparse 运行时定义，以使用 sparse 的“上下文”跟踪功能，应用于锁定。这些注释告诉 sparse 什么时候有锁，以及注释的函数的进入和退出。

`_must_hold` - 指定的锁在函数进入和退出时被持有。

`_acquires` - 指定的锁在函数退出时被持有，但在进入时不被持有。

`_releases` - 指定的锁在函数进入时被持有，但在退出时不被持有。

如果函数在不持有锁的情况下进入和退出，在函数内部以平衡的方式获取和释放锁，则不需要注释。上面的三个注释是针对 sparse 否则会报告上下文不平衡的情况。

## 获取 sparse 工具

你可以从 Sparse 的主页获取最新的发布版本：

<https://www.kernel.org/pub/software-devel/sparse/dist/>

或者，你也可以使用 git 克隆最新的 sparse 开发版本：

`git://git.kernel.org/pub/scm-devel/sparse/sparse.git`

一旦你下载了源码，只要以普通用户身份运行：

`make make install`

如果是标准的用户，它将会被自动安装到你的 `~/bin` 目录下。

## 使用 sparse 工具

用“`make C=1`”命令来编译内核，会对所有重新编译的 C 文件使用 sparse 工具。或者使用“`make C=2`”命令，无论文件是否被重新编译都会对其使用 sparse 工具。如果你已经编译了内核，用后一种方式可以很快地检查整个源码树。

`make` 的可选变量 `CHECKFLAGS` 可以用来向 sparse 工具传递参数。编译系统会自动向 sparse 工具传递 `-Wbitwise` 参数。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:<alexs@kernel.org>)。

**Original** Documentation/dev-tools/gcov.rst

**Translator** 赵军奎 Bernard Zhao <bernard@vivo.com>

### 在 Linux 内核里使用 gcov 做代码覆盖率检查

gcov 分析核心支持在 Linux 内核中启用 GCC 的覆盖率测试工具 `gcov`，Linux 内核运行时的代码覆盖率数据会以 `gcov` 兼容的格式导出到“`gcov`”`debugfs` 目录中，可以通过 `gcov` 的 `-o` 选项（如下示例）获得指定文件的代码运行覆盖率统计数据（需要跳转到内核编译路径下并且要有 root 权限）：

```
# cd /tmp/linux-out  
# gcov -o /sys/kernel/debug/gcov/tmp/linux-out/kernel spinlock.c
```

这将在当前目录中创建带有执行计数注释的源代码文件。在获得这些统计文件后，可以使用图形化的 `gcov` 前端工具（比如 `lcov`），来实现自动化处理 Linux 内核的覆盖率运行数据，同时生成易于阅读的 HTML 格式文件。

可能的用途：

- 调试（用来判断每一行的代码是否已经运行过）
- 测试改进（如何修改测试代码，尽可能地覆盖到没有运行过的代码）
- 内核最小化配置（对于某一个选项配置，如果关联的代码从来没有运行过，是否还需要这个配置）

### 准备

内核打开如下配置：

```
CONFIG_DEBUG_FS=y  
CONFIG_GCOV_KERNEL=y
```

获取整个内核的覆盖率数据，还需要打开：

```
CONFIG_GCOV_PROFILE_ALL=y
```

需要注意的是，整个内核开启覆盖率统计会造成内核镜像文件尺寸的增大，同时内核运行也会变慢一些。另外，并不是所有的架构都支持整个内核开启覆盖率统计。

代码运行覆盖率数据只在 `debugfs` 挂载完成后才可以访问：

```
mount -t debugfs none /sys/kernel/debug
```

## 定制化

如果要单独针对某一个路径或者文件进行代码覆盖率统计，可以在内核相应路径的 Makefile 中增加如下的配置：

- 单独统计单个文件（例如 main.o）：

```
GCOV_PROFILE_main.o := y
```

- 单独统计某一个路径：

```
GCOV_PROFILE := y
```

如果要在整个内核的覆盖率统计（开启 CONFIG\_GCOV\_PROFILE\_ALL）中单独排除某一个文件或者路径，可以使用如下的方法：

```
GCOV_PROFILE_main.o := n
```

和：

```
GCOV_PROFILE := n
```

此机制仅支持链接到内核镜像或编译为内核模块的文件。

## 相关文件

gcov 功能需要在 debugfs 中创建如下文件：

**/sys/kernel/debug/gcov** gcov 相关功能的根路径

**/sys/kernel/debug/gcov/reset** 全局复位文件：向该文件写入数据后会将所有的 gcov 统计数据清 0

**/sys/kernel/debug/gcov/path/to/compile/dir/file.gcda**

gcov 工具可以识别的覆盖率统计数据文件，向该文件写入数据后 会将本文件的 gcov 统计数据清 0

**/sys/kernel/debug/gcov/path/to/compile/dir/file.gcno** gcov 工具需要的软连接文件（指向编译时生成的信息统计文件），这个文件是在 gcc 编译时如果配置了选项 -ftest-coverage 时生成的。

## 针对模块的统计

内核中的模块会动态的加载和卸载，模块卸载时对应的数据会被清除掉。gcov 提供了一种机制，通过保留相关数据的副本 来收集这部分卸载模块的覆盖率数据。模块卸载后这些备份数据在 debugfs 中会继续存在。一旦这个模块重新加载，模块关联的运行统计会被初始化成 debugfs 中备份的数据。

可以通过对内核参数 gcov\_persist 的修改来停用 gcov 对模块的备份机制：

```
gcov_persist = 0
```

在运行时，用户还可以通过写入模块的数据文件或者写入 gcov 复位文件来丢弃已卸载模块的数据。

### 编译机和测试机分离

gcov 的内核分析插桩支持内核的编译和运行是在同一台机器上，也可以编译和运行是在不同的机器上。如果内核编译和运行是不同的机器，那么需要额外的准备工作，这取决于 gcov 工具是在哪里使用的：

#### a) 若 gcov 运行在测试机上

测试机上面 gcov 工具的版本必须要跟内核编译机器使用的 gcc 版本相兼容，同时下面的文件要从编译机拷贝到测试机上：

##### 从源代码中：

- 所有的 C 文件和头文件

##### 从编译目录中：

- 所有的 C 文件和头文件
- 所有的.gcda 文件和.gcno 文件
- 所有目录的链接

特别需要注意，测试机器上面的目录结构跟编译机器上面的目录机构必须完全一致。如果文件是软链接，需要替换成真正的目录文件（这是由 make 的当前工作目录变量 CURDIR 引起的）。

#### b) 若 gcov 运行在编译机上

测试用例运行结束后，如下的文件需要从测试机中拷贝到编译机上：

##### 从 sysfs 中的 gcov 目录中：

- 所有的.gcda 文件
- 所有的.gcno 文件软链接

这些文件可以拷贝到编译机的任意目录下，gcov 使用-o 选项指定拷贝的目录。

比如一个示例的目录结构如下：

```
/tmp/linux:    内核源码目录  
/tmp/out:      内核编译文件路径 (make O= 指定)  
/tmp/coverage: 从测试机器上面拷贝的数据文件路径  
  
[user@build] cd /tmp/out  
[user@build] gcov -o /tmp/coverage/tmp/out/init main.c
```

## 关于编译器的注意事项

GCC 和 LLVM gcov 工具不一定兼容。如果编译器是 GCC，使用 `gcov` 来处理.gcno 和.gcda 文件，如果是 Clang 编译器，则使用 `llvm-cov`。

GCC 和 Clang gcov 之间的版本差异由 Kconfig 处理的。kconfig 会根据编译工具链的检查自动选择合适的 gcov 格式。

## 问题定位

### 可能出现的问题 1 编译到链接阶段报错终止

**问题原因** 分析标志指定在了源文件但是没有链接到主内核，或者客制化了链接程序

**解决方法** 通过在相应的 Makefile 中使用 `GCOV_PROFILE := n` 或者 `GCOV_PROFILE_basename.o := n` 来将链接报错的文件排除掉

### 可能出现的问题 2 从 sysfs 复制的文件显示为空或不完整

**问题原因** 由于 `seq_file` 的工作方式，某些工具（例如 `cp` 或 `tar`）可能无法正确地从 sysfs 复制文件。

**解决方法** 使用 `cat` 读取 .gcda 文件，使用 `cp -d` 复制链接，或者使用附录 B 中所示的机制。

## 附录 A: collect\_on\_build.sh

用于在编译机上收集覆盖率元文件的示例脚本（见编译机和测试机分离 a.）

```
#!/bin/bash

KSRC=$1
KOBJ=$2
DEST=$3

if [ -z "$KSRC" ] || [ -z "$KOBJ" ] || [ -z "$DEST" ]; then
    echo "Usage: $0 <ksrc directory> <kobj directory> <output.tar.gz>" >&2
    exit 1
fi

KSRC=$(cd $KSRC; printf "all:\n\t@echo \$\{CURDIR\}\n" | make -f -)
KOBJ=$(cd $KOBJ; printf "all:\n\t@echo \$\{CURDIR\}\n" | make -f -)

find $KSRC $KOBJ \( -name '*.gcno' -o -name '.*.[ch]' -o -type l \| \) -a \
-perm /u+r,g+r | tar cfz $DEST -P -T -

if [ $? -eq 0 ] ; then
    echo "$DEST successfully created, copy to test system and unpack with:"
    echo " tar xzf $DEST -P"
else

```

```
echo "Could not create file $DEST"  
fi
```

### 附录 B: collect\_on\_test.sh

用于在测试机上收集覆盖率数据文件的示例脚本（见编译机和测试机分离 b.）

```
#!/bin/bash -e  
  
DEST=$1  
GCDA=/sys/kernel/debug/gcov  
  
if [ -z "$DEST" ] ; then  
    echo "Usage: $0 <output.tar.gz>" >&2  
    exit 1  
fi  
  
TEMPDIR=$(mktemp -d)  
echo Collecting data..  
find $GCDA -type d -exec mkdir -p $TEMPDIR/\{\}\; ;  
find $GCDA -name '*.gcda' -exec sh -c 'cat < $0 > '$TEMPDIR'/$0' {} \;  
find $GCDA -name '*.gcno' -exec sh -c 'cp -d $0 '$TEMPDIR'/$0' {} \;  
tar czf $DEST -C $TEMPDIR sys  
rm -rf $TEMPDIR  
  
echo "$DEST successfully created, copy to build system and unpack with:"  
echo " tar xzf $DEST"
```

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

Original Documentation/dev-tools/kasan.rst

Translator 万家兵 Wan Jiabing <[waniabing@vivo.com](mailto:waniabing@vivo.com)>

## 内核地址消毒剂 (KASAN)

### 概述

KernelAddressSANitizer(KASAN) 是一种动态内存安全错误检测工具，主要功能是检查内存越界访问和使用已释放内存的问题。KASAN 有三种模式：

1. 通用 KASAN (与用户空间的 ASan 类似)
2. 基于软件标签的 KASAN (与用户空间的 HWASan 类似)
3. 基于硬件标签的 KASAN (基于硬件内存标签)

由于通用 KASAN 的内存开销较大，通用 KASAN 主要用于调试。基于软件标签的 KASAN 可用于 dogfood 测试，因为它具有较低的内存开销，并允许将其用于实际工作量。基于硬件标签的 KASAN 具有较低的内存和性能开销，因此可用于生产。同时可用于检测现场内存问题或作为安全缓解措施。

软件 KASAN 模式 (#1 和 #2) 使用编译时工具在每次内存访问之前插入有效性检查，因此需要一个支持它的编译器版本。

通用 KASAN 在 GCC 和 Clang 受支持。GCC 需要 8.3.0 或更高版本。任何受支持的 Clang 版本都是兼容的，但从 Clang 11 才开始支持检测全局变量的越界访问。

基于软件标签的 KASAN 模式仅在 Clang 中受支持。

硬件 KASAN 模式 (#3) 依赖硬件来执行检查，但仍需要支持内存标签指令的编译器版本。GCC 10+ 和 Clang 11+ 支持此模式。

两种软件 KASAN 模式都适用于 SLUB 和 SLAB 内存分配器，而基于硬件标签的 KASAN 目前仅支持 SLUB。

目前 x86\_64、arm、arm64、xtensa、s390、riscv 架构支持通用 KASAN 模式，仅 arm64 架构支持基于标签的 KASAN 模式。

### 用法

要启用 KASAN，请使用以下命令配置内核：

```
CONFIG_KASAN=y
```

同时在 CONFIG\_KASAN\_GENERIC (启用通用 KASAN 模式)，CONFIG\_KASAN\_SW\_TAGS (启用基于硬件标签的 KASAN 模式)，和 CONFIG\_KASAN\_HW\_TAGS (启用基于硬件标签的 KASAN 模式) 之间进行选择。

对于软件模式，还可以在 CONFIG\_KASAN\_OUTLINE 和 CONFIG\_KASAN\_INLINE 之间进行选择。outline 和 inline 是编译器插桩类型。前者产生较小的二进制文件，而后者快 1.1-2 倍。

要将受影响的 slab 对象的 alloc 和 free 堆栈跟踪包含到报告中，请启用 CONFIG\_STACKTRACE。要包括受影响物理页面的分配和释放堆栈跟踪的话，请启用 CONFIG\_PAGE\_OWNER 并使用 page\_owner=on 进行引导。

### 错误报告

典型的 KASAN 报告如下所示：

```
=====
BUG: KASAN: slab-out-of-bounds in kmalloc_oob_right+0xa8/0xbc [test_kasan]
Write of size 1 at addr ffff8801f44ec37b by task insmod/2760

CPU: 1 PID: 2760 Comm: insmod Not tainted 4.19.0-rc3+ #698
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.10.2-1 04/01/2014
Call Trace:
dump_stack+0x94/0xd8
print_address_description+0x73/0x280
kasan_report+0x144/0x187
__asan_report_store1_noabort+0x17/0x20
kmalloc_oob_right+0xa8/0xbc [test_kasan]
kmalloc_tests_init+0x16/0x700 [test_kasan]
do_one_initcall+0xa5/0x3ae
do_init_module+0x1b6/0x547
load_module+0x75df/0x8070
__do_sys_init_module+0x1c6/0x200
__x64_sys_init_module+0x6e/0xb0
do_syscall_64+0x9f/0x2c0
entry_SYSCALL_64_after_hwframe+0x44/0xa9
RIP: 0033:0x7f96443109da
RSP: 002b:00007ffcf0b51b08 EFLAGS: 00000202 ORIG_RAX: 00000000000000af
RAX: ffffffff7fffffd R BX: 00005dc3ee521a0 RCX: 00007f96443109da
RDX: 00007f96445cff88 R SI: 0000000000057a50 RDI: 00007f9644992000
RBP: 000055dc3ee510b0 R08: 0000000000000003 R09: 0000000000000000
R10: 00007f964430cd0a R11: 00000000000000202 R12: 00007f96445cff88
R13: 000055dc3ee51090 R14: 0000000000000000 R15: 0000000000000000

Allocated by task 2760:
save_stack+0x43/0xd0
kasan_kmalloc+0xa7/0xd0
kmem_cache_alloc_trace+0xe1/0x1b0
kmalloc_oob_right+0x56/0xbc [test_kasan]
kmalloc_tests_init+0x16/0x700 [test_kasan]
do_one_initcall+0xa5/0x3ae
do_init_module+0x1b6/0x547
load_module+0x75df/0x8070
__do_sys_init_module+0x1c6/0x200
__x64_sys_init_module+0x6e/0xb0
do_syscall_64+0x9f/0x2c0
entry_SYSCALL_64_after_hwframe+0x44/0xa9

Freed by task 815:
save_stack+0x43/0xd0
__kasan_slab_free+0x135/0x190
kasan_slab_free+0xe/0x10
kfree+0x93/0x1a0
```

```
umh_complete+0x6a/0xa0  
call_usermodehelper_exec_async+0x4c3/0x640  
ret_from_fork+0x35/0x40
```

The buggy address belongs to the object at ffff8801f44ec300  
which belongs to the cache kmalloc-128 of size 128

The buggy address is located 123 bytes inside of  
128-byte region [fffff8801f44ec300, fffff8801f44ec380)

The buggy address belongs to the page:

page:fffffea0007d13b00 count:1 mapcount:0 mapping:fffff8801f7001640 index:0x0

flags: 0x2000000000000100(sLab)

raw: 0200000000000100 ffffffea0007d11dc0 0000001a0000001a fffff8801f7001640

raw: 0000000000000000 0000000080150015 00000001ffffffffff 0000000000000000

page dumped because: kasan: had access detected

Memory state around the buggy address:

fffff8801f44ec200: fc fc fc fc fc fc fc fc fh fh fh fh fh fh fh fh

ffff8801f44ec200: fe  
ffff8801f44ec280: fb fb fb fb fb fb fb fc fc fc fc fc fc fc

>fffff8801f44ec300: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03

6

fffff8801f44ec380: fc fc fc fc fc fc fc fc fb fb fb fb fb fb fb fb fb

fffff8801f44ec380: fc fc

报告标题总结了发生的错误类型以及导致该错误的访问类型。紧随其后的是错误访问的堆栈跟踪、所访问内存分配位置的堆栈跟踪（对于访问了 slab 对象的情况）以及对象被释放的位置的堆栈跟踪（对于访问已释放内存对象的情况）。如果报告中包含任何堆栈跟踪，它们将显示在报告末尾。

最后，报告展示了访问地址周围的内存状态。在内部，KASAN 单独跟踪每个内存颗粒的内存状态，根据 KASAN 模式分为 8 或 16 个对齐字节。报告的内存状态部分中的每个数字都显示了围绕访问地址的其中一个内存颗粒的状态。

对于通用 KASAN，每个内存颗粒的大小为 8 个字节。每个颗粒的状态被编码在一个影子字节中。这 8 个字节可以是可访问的，部分访问的，已释放的或成为 Redzone 的一部分。KASAN 对每个影子字节使用以下编码:00 表示对应内存区域的所有 8 个字节都可以访问；数字 N ( $1 \leq N \leq 7$ ) 表示前 N 个字节可访问，其他 ( $8 - N$ ) 个字节不可访问；任何负值都表示无法访问整个 8 字节。KASAN 使用不同的负值来区分不同类型的不可访问内存，如 redzones 或已释放的内存（参见 mm/kasan/kasan.h）。

在上面的报告中 箭头指向影子字节 03 表示访问的地址是部分可访问的。

对于基于标签的 KASAN 模式，报告最后的部分显示了访问地址周围的内存标签（参考 [实施细则](#) 章节）。

请注意，KASAN 错误标题（如 `slab-out-of-bounds` 或 `use-after-free`）是尽量接近的：KASAN 根据其拥有的有限信息打印出最可能的错误类型。错误的实际类型可能会有所不同。

通用 KASAN 还报告两个辅助调用堆栈跟踪。这些堆栈跟踪指向代码中与对象交互但不直接出现在错误访问堆栈跟踪中的位置。目前，这包括 `call_rcu()` 和排队的工作队列。

## 启动参数

KASAN 受通用 `panic_on_warn` 命令行参数的影响。启用该功能后，KASAN 在打印错误报告后会引起内核恐慌。

默认情况下，KASAN 只为第一次无效内存访问打印错误报告。使用 `kasan_multi_shot`，KASAN 会针对每个无效访问打印报告。这有效地禁用了 KASAN 报告的 `panic_on_warn`。

基于硬件标签的 KASAN 模式（请参阅下面有关各种模式的部分）旨在在生产中用作安全缓解措施。因此，它支持允许禁用 KASAN 或控制其功能的引导参数。

- `kasan=off` 或 `=on` 控制 KASAN 是否启用（默认：`on`）。
- `kasan.mode=sync` 或 `=async` 控制 KASAN 是否配置为同步或异步执行模式（默认：`sync`）。同步模式：当标签检查错误发生时，立即检测到错误访问。异步模式：延迟错误访问检测。当标签检查错误发生时，信息存储在硬件中（在 arm64 的 TFSR\_EL1 寄存器中）。内核会定期检查硬件，并且仅在这些检查期间报告标签错误。
- `kasan.stacktrace=off` 或 `=on` 禁用或启用 alloc 和 free 堆栈跟踪收集（默认：`on`）。
- `kasan.fault=report` 或 `=panic` 控制是只打印 KASAN 报告还是同时使内核恐慌（默认：`report`）。即使启用了 `kasan_multi_shot`，也会发生内核恐慌。

## 实施细则

### 通用 KASAN

软件 KASAN 模式使用影子内存来记录每个内存字节是否可以安全访问，并使用编译时工具在每次内存访问之前插入影子内存检查。

通用 KASAN 将 1/8 的内核内存专用于其影子内存（16TB 以覆盖 x86\_64 上的 128TB），并使用具有比例和偏移量的直接映射将内存地址转换为其相应的影子地址。

这是将地址转换为其相应影子地址的函数：

```
static inline void *kasan_mem_to_shadow(const void *addr)
{
    return (void *)((unsigned long)addr >> KASAN_SHADOW_SCALE_SHIFT)
           + KASAN_SHADOW_OFFSET;
}
```

在这里 `KASAN_SHADOW_SCALE_SHIFT = 3`。

编译时工具用于插入内存访问检查。编译器在每次访问大小为 1、2、4、8 或 16 的内存之前插入函数调用（`__asan_load*(addr)`, `__asan_store*(addr)`）。这些函数通过检查相应的影子内存来检查内存访问是否有效。

使用 `inline` 插桩，编译器不进行函数调用，而是直接插入代码来检查影子内存。此选项显著地增大了内核体积，但与 `outline` 插桩内核相比，它提供了 x1.1-x2 的性能提升。

通用 KASAN 是唯一一种通过隔离延迟重新使用已释放对象的模式（参见 mm/kasan/quarantine.c 以了解实现）。

## 基于软件标签的 KASAN 模式

基于软件标签的 KASAN 使用软件内存标签方法来检查访问有效性。目前仅针对 arm64 架构实现。

基于软件标签的 KASAN 使用 arm64 CPU 的顶部字节忽略 (TBI) 特性在内核指针的顶部字节中存储一个指针标签。它使用影子内存来存储与每个 16 字节内存单元相关的内存标签（因此，它将内核内存的 1/16 专用于影子内存）。

在每次内存分配时，基于软件标签的 KASAN 都会生成一个随机标签，用这个标签标记分配的内存，并将相同的标签嵌入到返回的指针中。

基于软件标签的 KASAN 使用编译时工具在每次内存访问之前插入检查。这些检查确保正在访问的内存的标签等于用于访问该内存的指针的标签。如果标签不匹配，基于软件标签的 KASAN 会打印错误报告。

基于软件标签的 KASAN 也有两种插桩模式（outline，发出回调来检查内存访问；inline，执行内联的影子内存检查）。使用 outline 插桩模式，会从执行访问检查的函数打印错误报告。使用 inline 插桩，编译器会发出 brk 指令，并使用专用的 brk 处理程序来打印错误报告。

基于软件标签的 KASAN 使用 0xFF 作为匹配所有指针标签（不检查通过带有 0xFF 指针标签的指针进行的访问）。值 0xFE 当前保留用于标记已释放的内存区域。

基于软件标签的 KASAN 目前仅支持对 Slab 和 page\_alloc 内存进行标记。

## 基于硬件标签的 KASAN 模式

基于硬件标签的 KASAN 在概念上类似于软件模式，但它是使用硬件内存标签作为支持而不是编译器插桩和影子内存。

基于硬件标签的 KASAN 目前仅针对 arm64 架构实现，并且基于 ARMv8.5 指令集架构中引入的 arm64 内存标记扩展 (MTE) 和最高字节忽略 (TBI)。

特殊的 arm64 指令用于为每次内存分配指定内存标签。相同的标签被指定给指向这些分配的指针。在每次内存访问时，硬件确保正在访问的内存的标签等于用于访问该内存的指针的标签。如果标签不匹配，则会生成故障并打印报告。

基于硬件标签的 KASAN 使用 0xFF 作为匹配所有指针标签（不检查通过带有 0xFF 指针标签的指针进行的访问）。值 0xFE 当前保留用于标记已释放的内存区域。

基于硬件标签的 KASAN 目前仅支持对 Slab 和 page\_alloc 内存进行标记。

如果硬件不支持 MTE (ARMv8.5 之前)，则不会启用基于硬件标签的 KASAN。在这种情况下，所有 KASAN 引导参数都将被忽略。

请注意，启用 CONFIG\_KASAN\_HW\_TAGS 始终会导致启用内核中的 TBI。即使提供了 kasan.mode=off 或硬件不支持 MTE (但支持 TBI)。

基于硬件标签的 KASAN 只报告第一个发现的错误。之后，MTE 标签检查将被禁用。

### 影子内存

内核将内存映射到地址空间的几个不同部分。内核虚拟地址的范围很大：没有足够的真实内存来支持内核可以访问的每个地址的真实影子区域。因此，KASAN 只为地址空间的某些部分映射真实的影子。

### 默认行为

默认情况下，体系结构仅将实际内存映射到用于线性映射的阴影区域（以及可能的其他小区域）。对于所有其他区域——例如 `vmalloc` 和 `vmemmap` 空间——一个只读页面被映射到阴影区域上。这个只读的影子页面声明所有内存访问都是允许的。

这给模块带来了一个问题：它们不存在于线性映射中，而是存在于专用的模块空间中。通过连接模块分配器，KASAN 临时映射真实的影子内存以覆盖它们。例如，这允许检测对模块全局变量的无效访问。

这也造成了与 `VMAP_STACK` 的不兼容：如果堆栈位于 `vmalloc` 空间中，它将被分配只读页面的影子内存，并且内核在尝试为堆栈变量设置影子数据时会出错。

### `CONFIG_KASAN_VMALLOC`

使用 `CONFIG_KASAN_VMALLOC`，KASAN 可以以更大的内存使用为代价覆盖 `vmalloc` 空间。目前，这在 x86、riscv、s390 和 powerpc 上受支持。

这通过连接到 `vmalloc` 和 `vmap` 并动态分配真实的影子内存来支持映射。

`vmalloc` 空间中的大多数映射都很小，需要不到一整页的阴影空间。因此，为每个映射分配一个完整的影子页面将是一种浪费。此外，为了确保不同的映射使用不同的影子页面，映射必须与 `KASAN_GRANULE_SIZE * PAGE_SIZE` 对齐。

相反，KASAN 跨多个映射共享后备空间。当 `vmalloc` 空间中的映射使用影子区域的特定页面时，它会分配一个后备页面。此页面稍后可以由其他 `vmalloc` 映射共享。

KASAN 连接到 `vmap` 基础架构以懒清理未使用的影子内存。

为了避免交换映射的困难，KASAN 预测覆盖 `vmalloc` 空间的阴影区域部分将不会被早期的阴影页面覆盖，但是将不会被映射。这将需要更改特定于 `arch` 的代码。

这允许在 x86 上支持 `VMAP_STACK`，并且可以简化对没有固定模块区域的架构的支持。

## 对于开发者

### 忽略访问

软件 KASAN 模式使用编译器插桩来插入有效性检查。此类检测可能与内核的某些部分不兼容，因此需要禁用。

内核的其他部分可能会访问已分配对象的元数据。通常，KASAN 会检测并报告此类访问，但在某些情况下（例如，在内存分配器中），这些访问是有效的。

对于软件 KASAN 模式，要禁用特定文件或目录的检测，请将 `KASAN_SANITIZE` 添加到相应的内核 Makefile 中：

- 对于单个文件（例如，`main.o`）：

```
KASAN_SANITIZE_main.o := n
```

- 对于一个目录下的所有文件：

```
KASAN_SANITIZE := n
```

对于软件 KASAN 模式，要在每个函数的基础上禁用检测，请使用 KASAN 特定的 `__no_sanitize_address` 函数属性或通用的 `noinstr`。

请注意，禁用编译器插桩（基于每个文件或每个函数）会使 KASAN 忽略在软件 KASAN 模式的代码中直接发生的访问。当访问是间接发生的（通过调用检测函数）或使用没有编译器插桩的基于硬件标签的模式时，它没有帮助。

对于软件 KASAN 模式，要在当前任务的一部分内核代码中禁用 KASAN 报告，请使用 `kasan_disable_current()`/`kasan_enable_current()` 部分注释这部分代码。这也会禁用通过函数调用发生的间接访问的报告。

对于基于标签的 KASAN 模式（包括硬件模式），要禁用访问检查，请使用 `kasan_reset_tag()` 或 `page_kasan_tag_reset()`。请注意，通过 `page_kasan_tag_reset()` 临时禁用访问检查需要通过 `page_kasan_tag` / `page_kasan_tag_set` 保存和恢复每页 KASAN 标签。

## 测试

有一些 KASAN 测试可以验证 KASAN 是否正常工作并可以检测某些类型的内存损坏。测试由两部分组成：

1. 与 KUnit 测试框架集成的测试。使用 `CONFIG_KASAN_KUNIT_TEST` 启用。这些测试可以通过几种不同的方式自动运行和部分验证；请参阅下面的说明。
2. 与 KUnit 不兼容的测试。使用 `CONFIG_KASAN_MODULE_TEST` 启用并且只能作为模块运行。这些测试只能通过加载内核模块并检查内核日志以获取 KASAN 报告来手动验证。

如果检测到错误，每个 KUnit 兼容的 KASAN 测试都会打印多个 KASAN 报告之一，然后测试打印其编号和状态。

当测试通过:

```
ok 28 - kmalloc_double_kzfree
```

当由于 kmalloc 失败而导致测试失败时:

```
# kmalloc_large_oob_right: ASSERTION FAILED at lib/test_kasan.c:163
Expected ptr is not null, but is
not ok 4 - kmalloc_large_oob_right
```

当由于缺少 KASAN 报告而导致测试失败时:

```
# kmalloc_double_kzfree: EXPECTATION FAILED at lib/test_kasan.c:629
Expected kasan_data->report_expected == kasan_data->report_found, but
kasan_data->report_expected == 1
kasan_data->report_found == 0
not ok 28 - kmalloc_double_kzfree
```

最后打印所有 KASAN 测试的累积状态。成功:

```
ok 1 - kasan
```

或者, 如果其中一项测试失败:

```
not ok 1 - kasan
```

有几种方法可以运行与 KUnit 兼容的 KASAN 测试。

### 1. 可加载模块

启用 CONFIG\_KUNIT 后, KASAN-KUnit 测试可以构建为可加载模块, 并通过使用 insmod 或 modprobe 加载 test\_kasan.ko 来运行。

### 2. 内置

通过内置 CONFIG\_KUNIT, 也可以内置 KASAN-KUnit 测试。在这种情况下, 测试将在启动时作为后期初始化调用运行。

### 3. 使用 kunit\_tool

通过内置 CONFIG\_KUNIT 和 CONFIG\_KASAN\_KUNIT\_TEST, 还可以使用 kunit\_tool 以更易读的方式查看 KUnit 测试结果。这不会打印通过测试的 KASAN 报告。有关 kunit\_tool 更多最新信息, 请参阅 [KUnit 文档](#)。

## orphan

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解, 而不是作为一个分支。因此, 如果您对此文件有任何意见或更新, 请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题, 请联系该文件的译者, 或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/dev-tools/gdb-kernel-debugging.rst

**Translator** 高超 gao chao <[gaochao49@huawei.com](mailto:gaochao49@huawei.com)>

## 通过 gdb 调试内核和模块

Kgdb 内核调试器、QEMU 等虚拟机管理程序或基于 JTAG 的硬件接口, 支持在运行时使用 gdb 调试 Linux 内核及其模块。Gdb 提供了一个强大的 python 脚本接口, 内核也提供了一套辅助脚本以简化典型的内核调试步骤。本文档为如何启用和使用这些脚本提供了一个简要的教程。此教程基于 QEMU/KVM 虚拟机, 但文中示例也适用于其他 gdb stub。

### 环境配置要求

- gdb 7.2+ (推荐版本: 7.4+) 且开启 python 支持 (通常发行版上都已支持)

### 设置

- 创建一个 QEMU/KVM 的 linux 虚拟机 (详情请参考 [www.linux-kvm.org](http://www.linux-kvm.org) 和 [www.qemu.org](http://www.qemu.org) )。对于交叉开发, <https://landley.net/aboriginal/bin> 提供了一些镜像和工具链, 可以帮助搭建交叉开发环境。
- 编译内核时开启 CONFIG\_GDB\_SCRIPTS, 关闭 CONFIG\_DEBUG\_INFO\_REDUCED。如果架构支持 CONFIG\_FRAME\_POINTER, 请保持开启。
- 在 guest 环境上安装该内核。如有必要, 通过在内核 command line 中添加 “nokaslr” 来关闭 KASLR。此外, QEMU 允许通过-kernel、-append、-initrd 这些命令行选项直接启动内核。但这通常仅在不依赖内核模块时才有效。有关此模式的更多详细信息, 请参阅 QEMU 文档。在这种情况下, 如果架构支持 KASLR, 应该在禁用 CONFIG\_RANDOMIZE\_BASE 的情况下构建内核。
- 启用 QEMU/KVM 的 gdb stub, 可以通过如下方式实现
  - 在 VM 启动时, 通过在 QEMU 命令行中添加 “-s” 参数

或

- 在运行时通过从 QEMU 监视控制台发送 “gdbserver”
- 切换到/path/to/linux-build(内核源码编译) 目录
- 启动 gdb: gdb vmlinux

注意: 某些发行版可能会将 gdb 脚本的自动加载限制在已知的安全目录中。如果 gdb 报告拒绝加载 vmlinux-gdb.py (相关命令找不到), 请将:

```
add-auto-load-safe-path /path/to/linux-build
```

添加到 `~/.gdbinit`。更多详细信息, 请参阅 `gdb` 帮助信息。

- 连接到已启动的 guest 环境:

```
(gdb) target remote :1234
```

### 使用 Linux 提供的 gdb 脚本的示例

- 加载模块（以及主内核）符号:

```
(gdb) lx-symbols
Loading vmlinux
scanning for modules in /home/user/linux/build
loading @0xfffffffffa0020000: /home/user/linux/build/net/netfilter/xt_tcpudp.ko
loading @0xfffffffffa0016000: /home/user/linux/build/net/netfilter/xt_pktype.ko
loading @0xfffffffffa0002000: /home/user/linux/build/net/netfilter/xt_limit.ko
loading @0xfffffffffa00ca000: /home/user/linux/build/net/packet/af_packet.ko
loading @0xfffffffffa003c000: /home/user/linux/build/fs/fuse/fuse.ko
...
loading @0xfffffffffa0000000: /home/user/linux/build/drivers/ata/ata_generic.ko
```

- 对一些尚未加载的模块中的函数函数设置断点, 例如:

```
(gdb) b btrfs_init_sysfs
Function "btrfs_init_sysfs" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (btrfs_init_sysfs) pending.
```

- 继续执行:

```
(gdb) c
```

- 加载模块并且能观察到正在加载的符号以及断点命中:

```
loading @0xfffffffffa0034000: /home/user/linux/build/lib/libcrc32c.ko
loading @0xfffffffffa0050000: /home/user/linux/build/lib/lzo/lzo_compress.ko
loading @0xfffffffffa006e000: /home/user/linux/build/lib/zlib_deflate/zlib_deflate.ko
loading @0xfffffffffa01b1000: /home/user/linux/build/fs/btrfs/btrfs.ko

Breakpoint 1, btrfs_init_sysfs () at /home/user/linux/fs/btrfs/sysfs.c:36
36           btrfs_kset = kset_create_and_add("btrfs", NULL, fs_kobj);
```

- 查看内核的日志缓冲区:

```
(gdb) lx-dmesg
[    0.000000] Initializing cgroup subsys cpuset
```

```
[ 0.000000] Initializing cgroup subsys cpu
[ 0.000000] Linux version 3.8.0-rc4-dbg+ (...
[ 0.000000] Command Line: root=/dev/sda2 resume=/dev/sda1 vga=0x314
[ 0.000000] e820: BIOS-provided physical RAM map:
[ 0.000000] BIOS-e820: [mem 0x0000000000000000-0x0000000000009fbff] usable
[ 0.000000] BIOS-e820: [mem 0x0000000000009fc00-0x0000000000009ffff] reserved
....
```

- 查看当前 task struct 结构体的字段（仅 x86 和 arm64 支持）：

```
(gdb) p $lx_current().pid
$1 = 4998
(gdb) p $lx_current().comm
$2 = "modprobe\000\000\000\000\000\000"
```

- 对当前或指定的 CPU 使用 per-cpu 函数：

```
(gdb) p $lx_per_cpu("runqueues").nr_running
$3 = 1
(gdb) p $lx_per_cpu("runqueues", 2).nr_running
$4 = 0
```

- 使用 container\_of 查看更多 hrtimers 信息：

```
(gdb) set $next = $lx_per_cpu("hrtimer_bases").clock_base[0].active.next
(gdb) p *$container_of($next, "struct hrtimer", "node")
$5 = {
  node = {
    node = {
      __rb_parent_color = 18446612133355256072,
      rb_right = 0x0 <irq_stack_union>,
      rb_left = 0x0 <irq_stack_union>
    },
    expires = {
      tv64 = 1835268000000
    }
  },
  _softexpires = {
    tv64 = 1835268000000
  },
  function = 0xffffffff81078232 <tick_sched_timer>,
  base = 0xfffff88003fd0d6f0,
  state = 1,
  start_pid = 0,
  start_site = 0xffffffff81055c1f <hrtimer_start_range_ns+20>,
  start_comm = "swapper/2\000\000\000\000\000"
}
```

### 命令和辅助调试功能列表

命令和辅助调试功能可能会随着时间的推移而改进，此文显示的是初始版本的部分示例：

```
(gdb) apropos lx
function lx_current -- Return current task
function lx_module -- Find module by name and return the module variable
function lx_per_cpu -- Return per-cpu variable
function lx_task_by_pid -- Find Linux task by PID and return the task_struct variable
function lx_thread_info -- Calculate Linux thread_info from task variable
lx-dmesg -- Print Linux kernel log buffer
lx-lsmod -- List currently loaded modules
lx-symbols -- (Re-)load symbols of Linux kernel and currently loaded modules
```

可以通过“help <command-name>”或“help function <function-name>”命令获取指定命令或指定调试功能的更多详细信息。

Todolist:

- coccinelle
- kcov
- ubsan
- kmemleak
- kcsan
- kfence
- kgdb
- ksselftest
- kunit/index

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/doc-guide/index.rst

译者 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

## \* 如何编写内核文档

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/doc-guide/sphinx.rst

译者 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

## 简介

Linux 内核使用 [Sphinx](#) 来把 Documentation 下的 [reStructuredText](#) 文件转换成漂亮的文档。使用 make [htmldocs](#) 或 make [pdfdocs](#) 命令即可构建 HTML 或 PDF 格式的文档。生成的文档放在 Documentation/output 文件夹中。

reStructuredText 文件可能包含来自源文件的结构化文档注释或 kernel-doc 注释。通常它们用于描述代码的功能、类型和设计。kernel-doc 注释有一些特殊的结构和格式，但除此之外，它们还被作为 reStructuredText 处理。

最后，有成千上万的纯文本文档文件散布在 Documentation 里。随着时间推移，其中一些可能会转换为 reStructuredText，但其中大部分仍保持纯文本。

## 安装 Sphinx

Documentation/ 下的 ReST 文件现在使用 sphinx1.7 或更高版本构建。

这有一个脚本可以检查 Sphinx 的依赖项。更多详细信息见[检查 Sphinx 依赖项](#)。

大多数发行版都附带了 Sphinx，但是它的工具链比较脆弱，而且在您的机器上升级它或其他一些 Python 包导致文档构建中断的情况并不少见。

避免此情况的一种方法是使用与发行版附带的不同的版本。因此，建议使用 virtualenv-3 或 virtualenv 在虚拟环境中安装 Sphinx，具体取决于发行版如何打包 Python3。

## Note:

- 1) html 输出建议使用 RTD 主题。根据 Sphinx 版本的不同，它应该用 pip install sphinx\_rtd\_theme 单独安装。

- 2) 一些 ReST 页面包含数学表达式。由于 Sphinx 的工作方式，这些表达式是使用 LaTeX 编写的。它需要安装 amsfonts 和 amsmath 宏包，以便显示。

总之，如您要安装 Sphinx 2.4.4 版本，应执行：

```
$ virtualenv sphinx_2.4.4
$ . sphinx_2.4.4/bin/activate
(sphinx_2.4.4) $ pip install -r Documentation/sphinx/requirements.txt
```

在运行 `. sphinx_2.4.4/bin/activate` 之后，提示符将变化，以指示您正在使用新环境。如果您打开了一个新的 shell，那么在构建文档之前，您需要重新运行此命令以再次进入虚拟环境中。

## 图片输出

内核文档构建系统包含一个扩展，可以处理 GraphViz 和 SVG 格式的图像（参见[图形图片](#)）。

为了让它工作，您需要同时安装 GraphViz 和 ImageMagick 包。如果没有安装这些软件包，构建系统仍将构建文档，但不会在输出中包含任何图像。

## PDF 和 LaTeX 构建

目前只有 Sphinx 2.4 及更高版本才支持这种构建。

对于 PDF 和 LaTeX 输出，还需要 XeLaTeX 3.14159265 版本。（译注：此版本号真实存在）

根据发行版的不同，您可能还需要安装一系列 `texlive` 软件包，这些软件包提供了 XeLaTeX 工作所需的最小功能集。

## 检查 Sphinx 依赖项

这有一个脚本可以自动检查 Sphinx 依赖项。如果它认得您的发行版，还会提示您所用发行版的安装命令：

```
$ ./scripts/sphinx-pre-install
Checking if the needed tools for Fedora release 26 (Twenty Six) are available
Warning: better to also install "texlive-luatex85".
You should run:

    sudo dnf install -y texlive-luatex85
    /usr/bin/virtualenv sphinx_2.4.4
    . sphinx_2.4.4/bin/activate
    pip install -r Documentation/sphinx/requirements.txt
```

`Can't build as 1 mandatory dependency is missing at ./scripts/sphinx-pre-install line 468.`

默认情况下，它会检查 html 和 PDF 的所有依赖项，包括图像、数学表达式和 LaTeX 构建的需求，并假设将使用虚拟 Python 环境。html 构建所需的依赖项被认为是必需的，其他依赖项则是可选的。

它支持两个可选参数：

--no-pdf

禁用 PDF 检查；

--no-virtualenv

使用 Sphinx 的系统打包，而不是 Python 虚拟环境。

## Sphinx 构建

生成文档的常用方法是运行 `make htmldocs` 或 `make pdfdocs`。还有其它可用的格式：请参阅 `make help` 的文档部分。生成的文档放在 `Documentation/output` 下相应格式的子目录中。

要生成文档，显然必须安装 Sphinx (`sphinx-build`)。要让 HTML 输出更漂亮，可以使用 Read the Docs Sphinx 主题 (`sphinx_rtd_theme`)。对于 PDF 输出，您还需要 XeLaTeX 和来自 ImageMagick (<https://wwwimagemagick.org>) 的 `convert(1)`。所有这些软件在大多发行版中都可用或已打包。

要传递额外的选项给 Sphinx，可以使用 `make` 变量 `SPHINXOPTS`。例如，使用 `make SPHINXOPTS=-v htmldocs` 获得更详细的输出。

要删除生成的文档，请运行 `make cleandocs`。

## 编写文档

添加新文档很容易，只需：

1. 在 `Documentation` 下某处添加一个新的 `.rst` 文件。
2. 从 `Documentation/index.rst` 中的 Sphinx 主目录树 链接到它。

对于简单的文档（比如您现在正在阅读的文档），这通常已经足够好了，但是对于较大的文档，最好创建一个子目录（或者使用现有的子目录）。例如，图形子系统文档位于 `Documentation/gpu` 下，拆分为多个 `.rst` 文件，并具有从主目录链接来的单独索引 `index.rst`（有自己的目录树 `toctree`）。

请参阅 [Sphinx](#) 和 [reStructuredText](#) 的文档，以了解如何使用它们。特别是 [Sphinx reStructuredText 基础](#) 这是开始学习使用 `reStructuredText` 的好地方。还有一些 [Sphinx 特殊标记结构](#)。

## 内核文档的具体指南

这是一些内核文档的具体指南：

- 请不要过于痴迷转换格式到 `reStructuredText`。保持简单。在大多数情况下，文档应该是纯文本，格式应足够一致，以便可以转换为其他格式。
- 将现有文档转换为 `reStructuredText` 时，请尽量减少格式更改。
- 在转换文档时，还要更新内容，而不仅仅是格式。

- 请遵循标题修饰符的顺序：

1. = 文档标题，要有上线：

```
=====  
文档标题  
=====
```

2. = 章：

```
章标题  
=====
```

3. - 节：

```
节标题  
-----
```

4. ~ 小节：

```
小节标题  
~~~~~
```

尽管 RST 没有规定具体的顺序（“没有强加一个固定数量和顺序的节标题装饰风格，最终按照的顺序将是实际遇到的顺序。”），但是拥有一个通用级别的文档更容易遵循。

- 对于插入固定宽度的文本块（用于代码样例、用例等）：:: 用于语法高亮意义不大的内容，尤其是短代码段；.. code-block:: <language> 用于需要语法高亮的较长代码块。对于嵌入到文本中的简短代码片段，请使用 ``。

## C 域

**Sphinx C 域 (Domain)** (name c) 适用于 C API 文档。例如，函数原型：

```
.. c:function:: int ioctl( int fd, int request )
```

内核文档的 C 域有一些附加特性。例如，您可以使用诸如 open 或 ioctl 这样的通用名称重命名函数的引用名称：

```
.. c:function:: int ioctl( int fd, int request )  
  :name: VIDIOC_LOG_STATUS
```

函数名称（例如 ioctl）仍保留在输出中，但引用名称从 ioctl 变为 VIDIOC\_LOG\_STATUS。此函数的索引项也变为 VIDIOC\_LOG\_STATUS。

请注意，不需要使用 c:func: 生成函数文档的交叉引用。由于一些 Sphinx 扩展的神奇力量，如果给定函数名的索引项存在，文档构建系统会自动将对 function() 的引用转换为交叉引用。如果在内核文档中看到 c:func: 的用法，请删除它。

## 列表

我们建议使用 **列式表格式**。列式表格式是二级列表。与 ASCII 艺术相比，它们对文本文件的读者来说可能没有那么舒适。但其优点是易于创建或修改，而且修改的差异（diff）更有意义，因为差异仅限于修改的内容。

平铺表也是一个二级列表，类似于 **列式表**，但具有一些额外特性：

- **列范围**：使用 `cspan` 修饰，可以通过其他列扩展单元格
- **行范围**：使用 `rspan` 修饰，可以通过其他行扩展单元格
- 自动将表格行最右边的单元格扩展到该行右侧空缺的单元格上。若使用 `:fill-cells:` 选项，此行为可以从 `自动合并` 更改为 `自动插入`，自动插入（空）单元格，而不是扩展合并到最后一个单元格。

选项：

- `:header-rows: [int]` 标题行计数
- `:stub-columns: [int]` 标题列计数
- `:widths: [[int] [int] ...]` 列宽
- `:fill-cells:` 插入缺少的单元格，而不是自动合并缺少的单元格

修饰：

- `:cspan: [int]` 扩展列
- `:rspan: [int]` 扩展行

下面的例子演示了如何使用这些标记。分级列表的第一级是 表格行。表格行中只允许一个标记，即该 表格行中的单元格列表。`comments` (...) 和 `targets` 例外（例如引用 `:ref:`最后一行 <last_row_zh>` / 最后一行`）。

```
.. flat-table:: 表格标题
  :widths: 2 1 1 3

  * - 表头 列 1
    - 表头 列 2
    - 表头 列 3
    - 表头 列 4

  * - 行 1
    - 字段 1.1
    - 字段 1.2 (自动扩展)

  * - 行 2
    - 字段 2.1
    - :rspan:`1` :cspan:`1` 字段 2.2~3.3

  * .. _`last_row_zh`:
    - 行 3
```

渲染效果：

Table 3: 表格标题

表头列 1	表头列 2	表头列 3	表头列 4
行 1	字段 1.1	字段 1.2 (自动扩展)	
行 2	字段 2.1	字段 2.2~3.3	
行 3			

## 交叉引用

从一页文档到另一页文档的交叉引用可以通过简单地写出文件路径来完成，无特殊格式要求。路径可以是绝对路径或相对路径。绝对路径从“Documentation/”开始。例如，要交叉引用此页，以下写法皆可，取决于具体的文档目录（注意 .rst 扩展名是可选的）：

参见 Documentation/doc-guide/sphinx.rst 。此法始终可用。  
请查看 sphinx.rst ，仅在同级目录中有效。  
请阅读 ../sphinx.rst ，上级目录中的文件。

如果要使用相对路径，则需要使用 Sphinx 的 doc 修饰。例如，从同一目录引用此页的操作如下：

参见 :doc:`sphinx 文档的自定义链接文本 <sphinx>` .

对于大多数用例，前者是首选，因为它更干净，更适合阅读源文件的人。如果您遇到一个没有任何特殊作用的 :doc: 用法，请将其转换为文档路径。

有关交叉引用 kernel-doc 函数或类型的信息，请参阅 Documentation/doc-guide/kernel-doc.rst 。

## 图形图片

如果要添加图片，应该使用 kernel-figure 和 kernel-image 指令。例如，要插入具有可缩放图像格式的图形，请使用 SVG (SVG 图片示例)：

```
.. kernel-figure:: ../../doc-guide/svg_image.svg
:alt: 简易 SVG 图片
```

SVG 图片示例

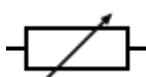


Fig. 1: SVG 图片示例

内核 figure (和 image) 指令支持 DOT 格式文件，请参阅

- DOT: <http://graphviz.org/pdf/dotguide.pdf>

- Graphviz: <http://www.graphviz.org/content/dot-language>

一个简单的例子 (**DOT** 示例) :

```
.. kernel-figure:: ../../doc-guide/hello.dot
:alt: 你好，世界
```

DOT 示例

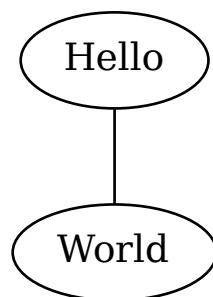


Fig. 2: DOT 示例

嵌入的渲染标记 (或语言), 如 Graphviz 的 **DOT** 由 `kernel-render` 指令提供:

```
.. kernel-render:: DOT
:alt: 有向图
:caption: 嵌入式 **DOT** (Graphviz) 代码

digraph foo {
    "五棵松" -> "国贸";
}
```

如何渲染取决于安装的工具。如果安装了 Graphviz, 您将看到一个矢量图像。否则, 原始标记将作为 文字块插入 ([嵌入式 DOT \(Graphviz\) 代码](#))。

`render` 指令包含 `figure` 指令中已知的所有选项, 以及选项 `caption`。如果 `caption` 有值, 则插入一个 `figure` 节点, 若无, 则插入一个 `image` 节点。如果您想引用它, 还需要一个 `caption` ([嵌入式 SVG 标记](#))。

嵌入式 **SVG**:

```
.. kernel-render:: SVG
:caption: 嵌入式 **SVG** 标记
:alt: 右上箭头

<?xml version="1.0" encoding="UTF-8"?>
<svg xmlns="http://www.w3.org/2000/svg" version="1.1" ...>
    ...

```

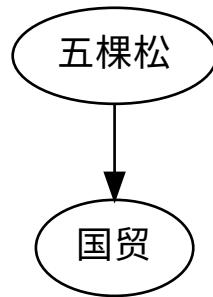


Fig. 3: 嵌入式 **DOT** (Graphviz) 代码

```
</svg>
```



Fig. 4: 嵌入式 **SVG** 标记

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

---

**Original** Documentation/doc-guide/kernel-doc.rst

译者 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

### 编写 kernel-doc 注释

Linux 内核源文件可以包含 kernel-doc 格式的结构化文档注释，用以描述代码的函数、类型和设计。将文档嵌入源文件更容易保持文档最新。

---

**Note:** 内核文档格式与 javadoc、gtk-doc 或 Doxygen 看似很相似，但由于历史原因，实际有着明显的不同。内核源包含成千上万个 kernel-doc 注释。请坚持遵循此处描述的风格。

---

---

**Note:** kernel-doc 无法包含 Rust 代码：请参考 Documentation/rust/docs.rst 。

---

从注释中提取 kernel-doc 结构，并从中生成适当的 Sphinx C 域 函数和带有锚点的类型描述。这些注释将被过滤以生成特殊 kernel-doc 高亮和交叉引用。详见下文。

使用 EXPORT\_SYMBOL 或 EXPORT\_SYMBOL\_GPL 导出到可加载模块的每个函数都应该有一个 kernel-doc 注释。模块使用的头文件中的函数和数据结构也应该有 kernel-doc 注释。

对于其他内核文件（未标记为 static）中外部可见的函数，提供 kernel-doc 格式的文档是一个很好的实践。我们也建议为私有（文件 static）程序提供 kernel-doc 格式的文档，以确保内核源代码布局的一致性。此建议优先级较低，由内核源文件的维护者自行决定。

## 如何格式化 kernel-doc 注释

kernel-doc 注释用 `/**` 作为开始标记。kernel-doc 工具将提取以这种方式标记的注释。注释其余部分的格式类似于一个普通的多行注释，左侧有一列星号，以 `*/` 行结束。

函数和类型的 kernel-doc 注释应该放在所描述的函数或类型之前，以便最大限度地提高更改代码的人同时更改文档的可能性。概述 kernel-doc 注释可以放在最顶部的任何地方。

用详细模式和不生成实际输出来运行 kernel-doc 工具，可以验证文档注释的格式是否正确。例如：

```
scripts/kernel-doc -v -none drivers/foo/bar.c
```

当请求执行额外的 gcc 检查时，内核构建将验证文档格式：

```
make W=n
```

## 函数文档

函数和函数式宏的 kernel-doc 注释的一般格式是：

```
/**
 * 函数名 () - 函数简要说明.
 * @ 参数 1: 描述第一个参数.
 * @ 参数 2: 描述第二个参数.
 *           可以为参数提供一段
 *           多行描述.
 *
 * 更详细的描述，进一步讨论函数 函数名 ()，这可能对使用或修改它的人有用.
 * 以空注释行开始，内部可以包含空注释行.
 *
 * 详细描述可以有多个段落.
 *
 * Context: 描述函数是否可以休眠，它需要、释放或期望持有什么锁.
```

```
*      可以写多行.  
* Return: 描述函数返回值.  
*  
* 返回值描述也可以有多个段落,  
* 并且应该放在注释块的末尾.  
*/
```

函数名后面的简短描述可以跨多行，并以参数描述、空注释行或注释块结尾结束。

### 函数参数

每个函数参数都应该按照顺序描述，紧跟在函数简要说明之后。不要在函数描述和参数之间，也不要在参数之间留空。

每个 @ 参数：描述可以跨多行。

---

**Note:** 如果 @ 参数描述有多行，则说明的续行应该从上一行的同一列开始：

```
* @ 参数: 较长说明  
*       的续行
```

或：

```
* @ 参数:  
*       较长说明  
*       的续行
```

---

如果函数的参数数目可变，则需用 kernel-doc 格式对其进行描述：

```
* @...: 描述
```

### 函数上下文

可调用函数的上下文应该在 Context 节中描述。此节应该包括函数是休眠的还是可以从中断上下文调用的，以及它需要什么锁、释放什么锁和期望它的调用者持有什么锁。

例如：

```
* Context: Any context.  
* Context: Any context. Takes and releases the RCU lock.  
* Context: Any context. Expects <lock> to be held by caller.  
* Context: Process context. May sleep if @gfp flags permit.  
* Context: Process context. Takes and releases <mutex>.  
* Context: Softirq or process context. Takes and releases <lock>, BH-safe.  
* Context: Interrupt context.
```

## 返回值

如有返回值，应在 Return 节中描述。

### Note:

- 1) 您提供的多行描述文本 不会识别换行符，因此如果您想将某些文本预格式化，如：

```
* Return:  
* 0 - OK  
* -EINVAL - invalid argument  
* -ENOMEM - out of memory
```

它们在最终文档中变成一行：

```
Return: 0 - OK -EINVAL - invalid argument -ENOMEM - out of memory
```

因此，为了在需要的地方换行，您需要使用 ReST 列表，例如：

```
* Return:  
* * 0 - OK to runtime suspend the device  
* * -EBUSY - Device should not be runtime suspended
```

- 2) 如果您提供的描述性文本中的行以某个后跟冒号的短语开头，则每一个这种短语都将被视为新的节标题，可能会产生意想不到的效果。

## 结构体、共用体、枚举类型文档

结构体 (struct)、共用体 (union)、枚举 (enum) 类型 kernel-doc 注释的一般格式为：

```
/**  
 * struct 结构体名 - 简要描述.  
 * @ 成员 1: 成员 1 描述.  
 * @ 成员 2: 成员 2 描述.  
 *          可以为成员提供  
 *          多行描述.  
 *  
 * 结构体的描述.  
 */
```

可以用 union 或 enum 替换上面示例中的 struct，以描述共用体或枚举。成员用于表示枚举中的元素或共用体成员。

结构体名称后面的简要说明可以跨多行，并以成员说明、空白注释行或注释块结尾结束。

### 成员

结构体、共用体和枚举的成员应以与函数参数相同的方式记录；它们后紧跟简短的描述，并且为多行。

在结构体或共用体描述中，可以使用 `private:` 和 `public:` 注释标签。`private:` 域内的字段不会列在生成的文档中。

`private:` 和 `public:` 标签必须紧跟在 `/*` 注释标记之后。可以选择是否在 `:` 和 `*/` 结束标记之间包含注释。

例子：

```
/**  
 * struct 张三 - 简短描述  
 * @a: 第一个成员  
 * @b: 第二个成员  
 * @d: 第三个成员  
 *  
 * 详细描述  
 */  
struct 张三 {  
    int a;  
    int b;  
/* private: 仅内部使用 */  
    int c;  
/* public: 下一个是公有的 */  
    int d;  
};
```

### 嵌套的结构体/共用体

嵌套的结构体/共用体可像这样记录：

```
/**  
 * struct nested_foo - a struct with nested unions and structs  
 * @memb1: first member of anonymous union/anonymous struct  
 * @memb2: second member of anonymous union/anonymous struct  
 * @memb3: third member of anonymous union/anonymous struct  
 * @memb4: fourth member of anonymous union/anonymous struct  
 * @bar: non-anonymous union  
 * @bar.st1: struct st1 inside @bar  
 * @bar.st2: struct st2 inside @bar  
 * @bar.st1.memb1: first member of struct st1 on union bar  
 * @bar.st1.memb2: second member of struct st1 on union bar  
 * @bar.st2.memb1: first member of struct st2 on union bar  
 * @bar.st2.memb2: second member of struct st2 on union bar  
 */  
struct nested_foo {  
    /* Anonymous union/struct*/  
    union {
```

```

struct {
    int memb1;
    int memb2;
};

struct {
    void *memb3;
    int memb4;
};

};

union {
    struct {
        int memb1;
        int memb2;
    } st1;
    struct {
        void *memb1;
        int memb2;
    } st2;
} bar;
};

```

**Note:**

- 1) 在记录嵌套结构体或共用体时，如果结构体/共用体 张三已命名，则其中的成员 李四应记录为 @ 张三. 李四：
- 2) 当嵌套结构体/共用体是匿名的时，其中的成员 李四应记录为 @ 李四：

**行间注释文档**

结构成员也可在定义时以行间注释形式记录。有两种样式，一种是单行注释，其中开始 `/**` 和结束 `*/` 位于同一行；另一种是多行注释，开头结尾各自位于一行，就像所有其他核心文档注释一样：

```

/**
 * struct 张三 - 简短描述.
 * @ 张三：成员张三.
 */
struct 张三 {
    int 张三;
    /**
     * @ 李四：成员李四.
     */
    int 李四;
    /**
     * @ 王五：成员王五.
     *
     * 此处，成员描述可以为好几段.

```

```

*/
int 王五;
union {
    /** @儿子: 单行描述. */
    int 儿子;
};
/** @赵六: 描述 @ 张三里面的结构体 @ 赵六 */
struct {
    /**
     * @赵六. 女儿: 描述 @ 张三. 赵六里面的 @ 女儿
     */
    int 女儿;
} 赵六;
};

```

## Typedef 文档

Typedef 的 kernel-doc 文档注释的一般格式为:

```

/**
 * typedef 类型名称 - 简短描述.
 *
 * 类型描述.
 */

```

还可以记录带有函数原型的 typedef:

```

/**
 * typedef 类型名称 - 简短描述.
 * @参数 1: 参数 1 的描述
 * @参数 2: 参数 2 的描述
 *
 * 类型描述.
 *
 * Context: 锁 (Locking) 上下文.
 * Return: 返回值的意义.
 */
typedef void (* 类型名称)(struct v4l2_ctrl * 参数 1, void * 参数 2);

```

## 高亮与交叉引用

在 kernel-doc 注释的描述文本中可以识别以下特殊模式，并将其转换为正确的 reStructuredText 标记和 Sphinx C 域引用。

**Attention:** 以下内容 仅在 kernel-doc 注释中识别，不会在普通的 reStructuredText 文档中识别。

**funcname()** 函数引用。

**@parameter** 函数参数的名称（未交叉引用，仅格式化）。

**%CONST** 常量的名称（未交叉引用，仅格式化）。

**``literal``** 预格式化文本块。输出将使用等距字体。

若你需要使用在 kernel-doc 脚本或 reStructuredText 中有特殊含义的字符，则此功能非常有用。

若你需要在函数描述中使用类似于 %ph 的东西，这特别有用。

**\$ENVVAR** 环境变量名称（未交叉引用，仅格式化）。

**&struct name** 结构体引用。

**&enum name** 枚举引用。

**&typedef name** Typedef 引用。

**&struct\_name->member or &struct\_name.member** 结构体或共用体成员引用。交叉引用将链接到结构体或共用体定义，而不是直接到成员。

**&name** 泛类型引用。请首选上面描述的完整引用方式。此法主要是为了可能未描述的注释。

## 从 reStructuredText 交叉引用

无需额外的语法来从 reStructuredText 文档交叉引用 kernel-do 注释中定义的函数和类型。只需以 () 结束函数名，并在类型之前写上 struct , union , enum 或 typedef 。例如：

```
See foo().  
See struct foo.  
See union bar.  
See enum baz.  
See typedef meh.
```

若要在交叉引用链接中使用自定义文本，可以通过以下语法进行：

```
See :c:func:`my custom link text for function foo <foo>`.  
See :c:type:`my custom link text for struct bar <bar>`.
```

有关更多详细信息，请参阅 Sphinx C 域 文档。

### 总述性文档注释

为了促进源代码和注释紧密联合，可以将 kernel-doc 文档块作为自由形式的注释，而不是函数、结构、联合、枚举或 `typedef` 的绑定 kernel-doc。例如，这可以用于解释驱动程序或库代码的操作理论。

这是通过使用带有节标题的 `DOC:` 节关键字来实现的。

总述或高层级文档注释的一般格式为：

```
/**  
 * DOC: Theory of Operation  
 *  
 * The whizbang foobar is a dilly of a gizmo. It can do whatever you  
 * want it to do, at any time. It reads your mind. Here's how it works.  
 *  
 * foo bar splat  
 *  
 * The only drawback to this gizmo is that it can sometimes damage  
 * hardware, software, or its subject(s).  
 */
```

`DOC:` 后面的标题用作源文件中的标题，但也用作提取文档注释的标识符。因此，文件中的标题必须是唯一的。

### 包含 kernel-doc 注释

文档注释可以被包含在任何使用专用 kernel-doc Sphinx 指令扩展的 reStructuredText 文档中。

kernel-doc 指令的格式如下：

```
.. kernel-doc:: source  
   :option:
```

*source* 是相对于内核源代码树的源文件路径。支持以下指令选项：

**export: [source-pattern …]** 包括 *source* 中使用 `EXPORT_SYMBOL` 或 `EXPORT_SYMBOL_GPL` 导出的所有函数的文档，无论是在 *source* 中还是在 *source-pattern* 指定的任何文件中。

当 kernel-doc 注释被放置在头文件中，而 `EXPORT_SYMBOL` 和 `EXPORT_SYMBOL_GPL` 位于函数定义旁边时，*source-pattern* 非常有用。

例子：

```
.. kernel-doc:: lib/bitmap.c  
   :export:  
  
.. kernel-doc:: include/net/mac80211.h  
   :export: net/mac80211/*.c
```

**internal: [source-pattern …]** 包括 *source* 中所有在 *source* 或 *source-pattern* 的任何文件中都没有使用 `EXPORT_SYMBOL` 或 `EXPORT_SYMBOL_GPL` 导出的函数和类型的文档。

例子:

```
.. kernel-doc:: drivers/gpu/drm/i915/intel_audio.c
:internal:
```

**identifiers: [ *function/type* … ]** 在 *source* 中包含每个 *function* 和 *type* 的文档。如果没有指定 *function*，则 *source* 中所有函数和类型的文档都将包含在内。

例子:

```
.. kernel-doc:: lib(bitmap.c
:identifiers: bitmap_parselist bitmap_parselist_user

.. kernel-doc:: lib/idr.c
:identifiers:
```

**no-identifiers: [ *function/type* … ]** 排除 *source* 中所有 *function* 和 *type* 的文档。

例子:

```
.. kernel-doc:: lib(bitmap.c
:no-identifiers: bitmap_parselist
```

**functions: [ *function/type* … ]** 这是“*identifiers*”指令的别名，已弃用。

**doc: *title*** 包含 *source* 中由 *title* 标题标识的 DOC: 文档段落。*title* 中允许空格；不要在 *title* 上加引号。*title* 仅用作段落的标识符，不包含在输出中。请确保在所附的 reStructuredText 文档中有适当的标题。

例子:

```
.. kernel-doc:: drivers/gpu/drm/i915/intel_audio.c
:doc: High Definition Audio over HDMI and Display Port
```

如果没有选项，kernel-doc 指令将包含源文件中的所有文档注释。

kernel-doc 扩展包含在内核源代码树中，位于 Documentation/sphinx/kerneldoc.py。在内部，它使用 scripts/kernel-doc 脚本从源代码中提取文档注释。

## 如何使用 kernel-doc 生成手册 (man) 页

如果您只想使用 kernel-doc 生成手册页，可以从内核 git 树这样做:

```
$ scripts/kernel-doc -man \
$(git grep -l '/\*\*/' -- :^Documentation :^tools) \
| scripts/split-man.pl /tmp/man
```

一些旧版本的 git 不支持路径排除语法的某些变体。以下命令之一可能适用于这些版本:

```
$ scripts/kernel-doc -man \
$(git grep -l '/\*\*' -- . 'Documentation' '!tools') \
| scripts/split-man.pl /tmp/man

$ scripts/kernel-doc -man \
$(git grep -l '/\*\*' -- . ":excludeDocumentation":exclude" ":(exclude)tools") \
| scripts/split-man.pl /tmp/man
```

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/doc-guide/parse-headers.rst

译者 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

### 包含用户空间 API 头文件

有时，为了描述用户空间 API 并在代码和文档之间生成交叉引用，需要包含头文件和示例 C 代码。为用户空间 API 文件添加交叉引用还有一个好处：如果在文档中找不到相应符号，Sphinx 将生成警告。这有助于保持用户空间 API 文档与内核更改同步。[parse\\_headers.pl](#) 提供了生成此类交叉引用的一种方法。在构建文档时，必须通过 Makefile 调用它。有关如何在内核树中使用它的示例，请参阅 Documentation/userspace-api/media/Makefile。

### parse\_headers.pl

#### 脚本名称

parse\_headers.pl——解析一个 C 文件，识别函数、结构体、枚举、定义并对 Sphinx 文档创建交叉引用。

## 用法概要

**parse\_headers.pl** [< 选项 >] <C 文件 > < 输出文件 > [< 例外文件 >]

< 选项 > 可以是: -debug, -help 或 -usage。

## 选项

### -debug

开启脚本详细模式，在调试时很有用。

### -usage

打印简短的帮助信息并退出。

### -help

打印更详细的帮助信息并退出。

## 说明

通过 C 头文件或源文件 (<C 文件 >) 中为描述 API 的文档编写的带交叉引用的.. 预格式化文本块将文件转换成重构文本 (RST)。它接受一个可选的 < 例外文件 >，其中描述了哪些元素将被忽略或指向非默认引用。输出被写入到 < 输出文件 >。

它能够识别定义、函数、结构体、typedef、枚举和枚举符号，并为它们创建交叉引用。它还能够区分用于指定 Linux ioctl 的 #define。

< 例外文件 > 包含两种类型的语句: **ignore** 或 **replace**.

**ignore** 标记的语法为:

#### ignore **type name**

The **ignore** 意味着它不会为类型为 **type** 的 **name** 符号生成交叉引用。

**replace** 标记的语法为:

#### replace **type name new\_value**

The **replace** 告诉它将为 **type** 类型的 **name** 符号生成交叉引用，但是它将使用 **new\_value** 来取代默认的替换规则。

这两种语句中，**type** 可以是以下任一项:

### ioctl

**ignore** 或 **replace** 语句应用于 ioctl 定义，如:

```
#define VIDIOC_DBG_S_REGISTER _IOW( 'V' , 79, struct v4l2_dbg_register)
```

### define

ignore 或 replace 语句应用于在 <C 文件 > 中找到的任何其他 #define 。

### typedef

ignore 和 replace 语句应用于 <C 文件 > 中的 typedef 语句。

### struct

ignore 和 replace 语句应用于 <C 文件 > 中的结构体名称语句。

### enum

ignore 和 replace 语句应用于 <C 文件 > 中的枚举名称语句。

### symbol

ignore 和 replace 语句应用于 <C 文件 > 中的枚举值名称语句。

replace 语句中, **new\_value** 会自动使用 **typedef**, **enum** 和 **struct** 类型的:c:type: 引用; 以及 **ioctl**, **define** 和 **symbol** 类型的:ref: 。引用的类型也可以在 replace 语句中显式定义。

## 示例

```
ignore define _VIDEODEV2_H
```

忽略 <C 文件 > 中的 #define \_VIDEODEV2\_H 。

```
ignore symbol PRIVATE
```

如下结构体:

```
enum foo { BAR1, BAR2, PRIVATE };
```

不会为 **PRIVATE** 生成交叉引用。

```
replace symbol BAR1 :c:type:`foo` replace symbol BAR2 :c:type:`foo`
```

如下结构体:

```
enum foo { BAR1, BAR2, PRIVATE };
```

它会让 BAR1 和 BAR2 枚举符号交叉引用 C 域中的 foo 符号。

## 缺陷

请向 Mauro Carvalho Chehab <[mchehab@kernel.org](mailto:mchehab@kernel.org)> 报告有关缺陷。

中文翻译问题请找中文翻译维护者。

## 版权

版权所有 (c) 2016 Mauro Carvalho Chehab <[mchehab+samsung@kernel.org](mailto:mchehab+samsung@kernel.org)>

许可证 GPLv2: GNU GPL version 2 <<https://gnu.org/licenses/gpl.html>>

这是自由软件：你可以自由地修改和重新发布它。在法律允许的范围内，**不提供任何保证**。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/doc-guide/contributing.rst

译者 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

## 如何帮助改进内核文档

在任何软件开发项目中，文档都是重要组成部分。好的文档有助于引入新的开发人员，并帮助已有的开发人员更有效地工作。如果缺少高质量的文档，大量的时间就会浪费在代码的逆向工程和犯本可避免的错误上。

不幸的是，内核的文档目前远远不能满足支持如此规模和重要性的项目的需要。

本指南适用于希望帮助改善这种状况的贡献者。内核文档的改进可以由开发者在不同的技能层级上进行；这也是一条相对简单可以帮助您了解内核过程并在社区中找到一席之地的路径。下面的大部分内容是文档维护人员列出的最迫切需要完成的任务。

## 文档待办事项列表

为了使我们的文档达到应有的水平，需要完成的任务数不胜数。此列表包含许多重要的项目，但还远远不够详尽；如果您知道改进文档的其他方法，请不要羞于启齿。

## 消除警告（WARNING）

文档构建目前出现了数量惊人的警告。虱子多了不痒，债多了不愁；大伙儿忽略了它们，他们永远不会注意到他们的工作增加了新的警告。因此，消除警告是文档待办事项列表中优先级最高的任务之一。这项任务本身相当简单，但必须以正确的方式进行，才能取得成功。

C 代码编译器发出的警告常常会被视为误报，从而导致出现了旨在让编译器闭嘴的补丁。文档构建中的警告几乎总是指向真正的问题；要消除这些警告，需要理解问题并从源头上解决问题。因此，修复文档警告的补丁不应在标题中直接写“修复警告”；它们应该指明真正修复的问题。

另一个重点是，文档警告常常由 C 代码里 kernel-doc 注释中的问题引起。虽然文档维护人员对收到这些修复补丁的副本表示感谢，但是文档树实际上通常并不适合接受这些补丁；它们应该被交给相关子系统的维护人员。

例如，在一次文档构建中，我几乎是随意选取了一对警告：

```
./drivers/devfreq/devfreq.c:1818: warning: bad line:  
  - Resource-managed devfreq_register_notifier()  
./drivers/devfreq/devfreq.c:1854: warning: bad line:  
  - Resource-managed devfreq_unregister_notifier()
```

(作了断行以便于阅读)

简单看一下上面给出的源文件，会发现几个 kernel-doc 注释，如下所示：

```
/**  
 * devm_devfreq_register_notifier()  
 *   - Resource-managed devfreq_register_notifier()  
 * @dev:      The devfreq user device. (parent of devfreq)  
 * @devfreq:   The devfreq object.  
 * @nb:       The notifier block to be unregistered.  
 * @list:     DEVFREQ_TRANSITION_NOTIFIER.  
 */
```

问题在于缺了一个“\*”，这不符合构建系统对 C 注释块的格式要求。此问题自 2016 年注释被添加以来一直存在——整整四年之久。修复它只需要添加丢失的星号。看一眼该文件的历史记录以了解主题行的常规格式是什么样，再使用 scripts/get\_maintainer.pl 来搞清谁应当收到此补丁。生成的补丁如下所示：

```
[PATCH] PM / devfreq: Fix two malformed kerneldoc comments
```

Two kerneldoc comments in devfreq.c fail to adhere to the required format,  
resulting in these doc-build warnings:

```
./drivers/devfreq/devfreq.c:1818: warning: bad line:  
  - Resource-managed devfreq_register_notifier()  
./drivers/devfreq/devfreq.c:1854: warning: bad line:  
  - Resource-managed devfreq_unregister_notifier()
```

Add a couple of missing asterisks and make kerneldoc a little happier.

Signed-off-by: Jonathan Corbet <corbet@lwn.net>

---

```
drivers/devfreq/devfreq.c | 4 +-  
1 file changed, 2 insertions(+), 2 deletions(-)
```

```
diff --git a/drivers/devfreq/devfreq.c b/drivers/devfreq/devfreq.c  
index 57f6944d65a6..00c9b80b3d33 100644  
--- a/drivers/devfreq/devfreq.c
```

```

+++ b/drivers/devfreq/devfreq.c
@@ -1814,7 +1814,7 @@ static void devm_devfreq_notifier_release(struct device *dev, void *res)

 /**
 * devm_devfreq_register_notifier()
- - Resource-managed devfreq_register_notifier()
+ - Resource-managed devfreq_register_notifier()
 * @dev:      The devfreq user device. (parent of devfreq)
 * @devfreq:   The devfreq object.
 * @nb:        The notifier block to be unregistered.
@@ -1850,7 +1850,7 @@ EXPORT_SYMBOL(devm_devfreq_register_notifier);

 /**
 * devm_devfreq_unregister_notifier()
- - Resource-managed devfreq_unregister_notifier()
+ - Resource-managed devfreq_unregister_notifier()
 * @dev:      The devfreq user device. (parent of devfreq)
 * @devfreq:   The devfreq object.
 * @nb:        The notifier block to be unregistered.
--
```

2.24.1

整个过程只花了几分钟。当然，我后来发现有人在另一个树中修复了它，这亮出了另一个教训：在深入研究问题之前，一定要检查 linux-next 树，看看问题是否已经修复。

其他修复可能需要更长的时间，尤其是那些与缺少文档的结构体成员或函数参数相关的修复。这种情况下，需要找出这些成员或参数的作用，并正确描述它们。总之，这种任务有时会有点乏味，但它非常重要。如果我们真的可以从文档构建中消除警告，那么我们就可以开始期望开发人员开始注意避免添加新的警告了。

## “迷失的” kernel-doc 注释

开发者被鼓励去为他们的代码写 kernel-doc 注释，但是许多注释从未被引入文档构建。这使得这些信息更难找到，例如使 Sphinx 无法生成指向该文档的链接。将 kernel-doc 指令添加到文档中以引入这些注释可以帮助社区获得为编写注释所做工作的全部价值。

`scripts/find-unused-docs.sh` 工具可以用来找到这些被忽略的评论。

请注意，将导出的函数和数据结构引入文档是最有价值的。许多子系统还具有供内部使用的 kernel-doc 注释；除非这些注释放在专门针对相关子系统开发人员的文档中，否则不应将其引入文档构建中。

### 修正错字

修复文档中的排版或格式错误是一种快速学习如何创建和发送修补程序的方法，也是一项有用的服务。我总是愿意接受这样的补丁。这也意味着，一旦你修复了一些这种错误，请考虑转移到更高级的任务，留下一些拼写错误给下一个初学者解决。

请注意，有些并不是拼写错误，不应该被“修复”：

- 内核文档中用美式和英式英语拼写皆可，没有必要互相倒换。
- 在内核文档中，没必要讨论句点后面应该跟一个还是两个空格的问题。其他一些有合理分歧的地方，比如“牛津逗号”，在这也是跑题的。

与对任何项目的任何补丁一样，请考虑您的更改是否真的让事情变得更好。

### “上古” 文档

一些内核文档是最新的、被维护的，并且非常有用，有些文件确并非如此。尘封、陈旧和不准确的文档可能会误导读者，并对我们的整个文档产生怀疑。任何解决这些问题的事情都是非常受欢迎的。

无论何时处理文档，请考虑它是否是最新的，是否需要更新，或者是否应该完全删除。您可以注意以下几个警告标志：

- 对 2.x 内核的引用
- 指向 SourceForge 存储库
- 历史记录除了拼写错误啥也没有持续几年
- 讨论 Git 之前时代的工作流

当然，最好的办法是更新文档，添加所需的任何信息。这样的工作通常需要与熟悉相关子系统的开发人员合作。当有人善意地询问开发人员，并听取他们的回答然后采取行动时，开发人员通常更愿意与这些致力于改进文档的人员合作。

有些文档已经没希望了；例如，我们偶尔会发现引用了很久以前从内核中删除的代码的文档。删除过时的文档会碰见令人惊讶的阻力，但我们无论如何都应该这样做。文档中多余的粗枝大叶对任何人都没有帮助。

如果一个严重过时的文档中可能有一些有用的信息，而您又无法更新它，那么最好在开头添加一个警告。建议使用以下文本：

```
.. warning ::  
    This document is outdated and in need of attention. Please use  
    this information with caution, and please consider sending patches  
    to update it.
```

这样的话，至少我们长期受苦的读者会得到文件可能会把他们引入歧途的警告。

## 文档一致性

这里的老前辈们会记得上世纪 90 年代出现在书架上的 Linux 书籍，它们只是从网上不同位置搜来的文档文件的集合。在此之后，(大部分) 这些书都得到了改进，但是内核的文档仍然主要是建立在这种模型上。它有数千个文件，几乎每个文件都是与其他文件相独立编写的。我们没有一个连贯的内核文档；只有数千个独立的文档。

我们一直试图通过编纂一套“书籍”来改善这种情况，以为特定读者提供成套文档。这包括：

- [Linux 内核用户和管理员指南](#)
- Documentation/core-api/index.rst
- Documentation/driver-api/index.rst
- Documentation/userspace-api/index.rst

以及文档本身这本“书”。

将文档移到适当的书中是一项重要的任务，需要继续进行。不过这项工作还是有一些挑战性。移动文档会给处理这些文档的人带来短期的痛苦；他们对这些更改无甚热情也是可以理解的。通常情况下，可以将一个文档移动一下；不过我们真的不想一直移动它们。

即使所有文件都在正确的位置，我们也只是把一大堆文件变成一群小堆文件。试图将所有这些文件组合成一个整体的工作尚未开始。如果你对如何在这方面取得进展有好的想法，我们将很高兴了解。

## 样式表 (Stylesheet) 改进

随着 Sphinx 的采用，我们得到了比以前更好的 HTML 输出。但它仍然需要很大的改进；Donald Knuth 和 Edward Tufte 可能是映像平平的。这需要调整我们的样式表，以创建更具排版效果、可访问性和可读性的输出。

请注意：如果你承担这个任务，你将进入经典的两难领域。即使是相对明显的变化，也会有很多意见和讨论。唉，这就是我们生活的世界的本质。

## 无 LaTeX 的 PDF 构建

对于拥有大量时间和 Python 技能的人来说，这绝对是一项不平凡的任务。Sphinx 工具链相对较小且包含良好；很容易添加到开发系统中。但是构建 PDF 或 EPUB 输出需要安装 LaTeX，它绝对称不上小或包含良好的。消除 Latex 将是一件很好的事情。

最初是希望使用 `rst2pdf` 工具来生成 PDF，但结果发现无法胜任这项任务。不过 `rst2pdf` 的开发工作最近似乎又有了起色，这是个充满希望的迹象。如果有开发人员愿意与该项目合作，使 `rst2pdf` 可与内核文档构建一起工作，大家会非常感激。

### 编写更多文档

当然，内核中许多部分的文档严重不足。如果您有编写一个特定内核子系统文档的相应知识并愿意这样做，请不要犹豫，编写并向内核贡献结果吧！数不清的内核开发人员和用户会感谢你。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

---

**Original** Documentation/doc-guide/maintainer-profile.rst

译者 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

### 文档子系统维护人员条目概述

文档“子系统”是内核文档和相关基础设施的中心协调点。它涵盖了 Documentation/ 下的文件层级 (Documentation/devicetree 除外)、scripts/ 下的各种实用程序，并且在某些情况下的也包括 LICENSES/。

不过值得注意的是，这个子系统的边界比通常更加模糊。许多其他子系统维护人员需要保持对 Documentation/ 某些部分的控制，以便于可以更自由地做更改。除此之外，许多内核文档都以 kernel-doc 注释的形式出现在源代码中；这些注释通常（但不总是）由相关的子系统维护人员维护。

文档子系统的邮件列表是 <[linux-doc@vger.kernel.org](mailto:linux-doc@vger.kernel.org)>。补丁应尽量针对 docs-next 树。

### 提交检查单补遗

在进行文档更改时，您应当构建文档以测试，并确保没有引入新的错误或警告。生成 HTML 文档并查看结果将有助于避免对文档渲染结果的不必要争执。

### 开发周期的关键节点

补丁可以随时发送，但在合并窗口期间，响应将比通常慢。文档树往往在合并窗口打开之前关闭得比较晚，因为文档补丁导致回归的风险很小。

## 审阅节奏

我（译注：指 Jonathan Corbet <corbet@lwn.net>）是文档子系统的唯一维护者，我在自己的时间里完成这项工作，所以对补丁的响应有时会很慢。当补丁被合并时（或当我决定拒绝合并补丁时），我都会发送通知。如果您在发送补丁后一周内没有收到回复，请不要犹豫，发送提醒就好。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/kernel-hacking/index.rst

译者 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

## \* 内核骇客指南

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/kernel-hacking/hacking.rst

译者 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

## 内核骇客指北

作者 Rusty Russell

### 引言

欢迎咱优雅的读者们来阅读 Rusty 的非常不靠谱的 Linux 内核骇客（Hacking）指南。本文描述了内核代码的常见例程和一般要求：其目标是引导有经验的 C 程序员入门 Linux 内核开发。我回避了实现细节：这是代码要做的，也忽略了很多有用的例程。

在你读这篇文章之前，请理解我从来没有想过要写这篇文章，因为我的资历太低了；但我一直想读这样的文章，自己写是唯一的方法。我希望它能成长为一个最佳实践、通用起点和其他信息的汇编。

### 玩家

在任何时候，系统中的每个 CPU 都可以：

- 与任何进程无关，服务于硬件中断；
- 与任何进程无关，服务于软件中断（softirq）或子任务（tasklet）；
- 运行于内核空间中，与进程（用户上下文）相关联；
- 在用户空间中运行进程。

它们之间有优先级顺序。最下面的两个可以互相抢占，但上面为严格的层次结构：每个层级只能被上方的抢占。例如，当一个软中断在 CPU 上运行时，没有其他软中断会抢占它，但是硬件中断可以抢占它。不过，系统中的任何其他 CPU 都是独立执行的。

我们将会看到许多方法，用户上下文可以阻止中断，从而成为真正的不可抢占。

### 用户上下文

用户上下文是指当您从系统调用或其他陷阱进入时：就像用户空间一样，您可以被更重要的任务和中断抢占。您可以通过调用 `schedule()` 进行睡眠。

---

**Note:** 在模块加载和卸载以及块设备层上的操作时，你始终处于用户上下文中。

---

在用户上下文中，当前 `current` 指针（指示我们当前正在执行的任务）是有效的，且 `in_interrupt()` (`include/linux/preempt.h`) 值为非（false）。

**Warning:** 请注意，如果您禁用了抢占或软中断（见下文），`in_interrupt()` 会返回假阳性。

## 硬件中断 (Hard IRQs)

像定时器、网卡和键盘等都是可能在任意时刻产生中断的真实硬件。内核运行中断处理程序，为硬件提供服务。内核确保处理程序永远不会重入：如果相同的中断到达，它将被排队（或丢弃）。因为它会关闭中断，所以处理程序必须很快：通常它只是确认中断，标记一个“软件中断”以执行并退出。

您可以通过 `in_hardirq()` 返回真来判断您处于硬件中断状态。

**Warning:** 请注意，如果中断被禁用，这将返回假阳性（见下文）。

## 软件中断上下文：软中断 (Softirqs) 与子任务 (Tasklets)

当系统调用即将返回用户空间或硬件中断处理程序退出时，任何标记为挂起（通常通过硬件中断）的“软件中断”将运行 (`kernel/softirq.c`)。

此处完成了许多真正的中断处理工作。在向 SMP 过渡的早期，只有“bottom halves 下半部”(BHs) 机制，无法利用多个 CPU 的优势。在从那些一团糟的就电脑切换过来后不久，我们放弃了这个限制，转而使用“软中断”。

`include/linux/interrupt.h` 列出了不同的软中断。定时器软中断是一个非常重要的软中断 (`include/linux/timer.h`)：您可以注册它以在给定时间后为您调用函数。

软中断通常是一个很难处理的问题，因为同一个软中断将同时在多个 CPU 上运行。因此，子任务 (`include/linux/interrupt.h`) 更常用：它们是动态可注册的（意味着您可以拥有任意数量），并且它们还保证任何子任务都只能在一个 CPU 上运行，不同的子任务也可以同时运行。

**Warning:** “tasklet”这个名字是误导性的：它们与“任务”无关，可能更多与当时阿列克谢·库兹涅佐夫享用的糟糕伏特加有关。

你可以使用 `in_softirq()` 宏 (`include/linux/preempt.h`) 来确认是否处于软中断（或子任务）中。

**Warning:** 注意，如果持有 `bottom half lock` 锁，这将返回假阳性。

### 一些基本规则

**缺少内存保护** 如果你损坏了内存，无论是在用户上下文还是中断上下文中，整个机器都会崩溃。你确定你不能在用户空间里做你想做的事吗？

**缺少浮点或 MMX** FPU 上下文不会被保存；即使在用户上下文中，FPU 状态也可能与当前进程不一致：您会弄乱某些用户进程的 FPU 状态。如果真的要这样做，就必须显式地保存/恢复完整的 FPU 状态（并避免上下文切换）。这通常不是个好主意；请优先用定点算法。

**严格的堆栈限制** 对于大多数 32 位体系结构，根据配置选项的不同内核堆栈大约为 3K 到 6K；对于大多数 64 位机器，内核堆栈大约为 14K，并且经常与中断共享，因此你无法使用全部。应避免深度递归和栈上的巨型本地数组（用动态分配它们来代替）。

**Linux 内核是可移植的** 就这样吧。您的代码应该是纯 64 位的，并且不依赖于字节序（endianness）。您还应该尽量减少 CPU 特定的东西，例如内联汇编（inline assembly）应该被干净地封装和最小化以便于移植。一般来说，它应该局限于内核树中有体系结构依赖的部分。

### 输入输出控制（ioctls）：避免编写新的系统调用

系统调用（system call）通常看起来像这样：

```
asmlinkage long sys_mycall(int arg)
{
    return 0;
}
```

首先，在大多数情况下，您无需创建新的系统调用。创建一个字符设备并为其实现适当的输入输出控制（ioctls）。这比系统调用灵活得多，不必写进每个体系结构的 `include/asm/unistd.h` 和 `arch/kernel/entry.S` 文件里，而且更容易被 Linus 接受。

如果您的程序所做的只是读取或写入一些参数，请考虑实现 `sysfs()` 接口。

在输入输出控制中，您处于进程的用户上下文。出现错误时，返回一个负的错误参数（`errno`，请参阅 `include/uapi/asm-generic/errno-base.h`、`include/uapi/asm-generic/errno.h` 和 `include/linux/errno.h`），否则返回 0。

在睡眠之后，您应该检查是否出现了信号：Unix/Linux 处理信号的方法是暂时退出系统调用，并返回 `-ERESTARTSYS` 错误。系统调用入口代码将切换回用户上下文，处理信号处理程序，然后系统调用将重新启动（除非用户禁用了该功能）。因此，您应该准备好处理重新启动，例如若您处理某些数据结构到一半。

```
if (signal_pending(current))
    return -ERESTARTSYS;
```

如果你要做更长时间的计算：优先考虑用户空间。如果你真的想在内核中做这件事，你应该定期检查你是否需要让出 CPU（请记得每个 CPU 都有协作多任务）。习惯用法：

```
cond_resched(); /* Will sleep */
```

接口设计的小注释：UNIX 系统调用的格言是“提供机制而不是策略 Provide mechanism not policy”。

## 死锁的“配方”

您不能调用任何可能睡眠的程序，除非：

- 您处于用户上下文中。
- 你未拥有任何自旋锁。
- 您已经启用中断（实际上，Andi Kleen 说调度代码将为您启用它们，但这可能不是您想要的）。

注意，有些函数可能隐式地睡眠：常见的是用户空间访问函数 (\*\_user) 和没有 GFP\_ATOMIC 的内存分配函数。

您应该始终打开 CONFIG\_DEBUG\_ATOMIC\_SLEEP 项来编译内核，如果您违反这些规则，它将警告您。如果你真的违反了规则，你最终会锁住你的电脑。

真的会这样。

## 常用函数/程序

### printk()

定义于 include/linux/printk.h

printk() 将内核消息提供给控制台、dmesg 和 syslog 守护进程。它对于调试和报告错误很有用，并且可以在中断上下文中使用，但是使用时要小心：如果机器的控制台中充斥着 printk 消息则会无法使用。它使用与 ANSI C printf 基本兼容的格式字符串，并通过 C 字符串串联为其提供第一个“优先”参数：

```
printk(KERN_INFO "i = %u\n", i);
```

参见 include/linux/kern\_levels.h；了解其他 KERN\_ 值；syslog 将这些值解释为级别。特殊用法：打印 IP 地址使用：

```
_be32 ipaddress;
printk(KERN_INFO "my ip: %pI4\n", &ipaddress);
```

printk() 内部使用的 1K 缓冲区，不捕获溢出。请确保足够使用。

---

**Note:** 当您开始在用户程序中将 printf 打成 printk 时，就知道自己是真正的内核程序员了：)

---

**Note:** 另一个注释：最初的 unix 第六版源代码在其 printf 函数的顶部有一个注释：“printf 不应该用于叽叽喳喳”。你也应该遵循此建议。

---

### `copy_to_user()` / `copy_from_user()` / `get_user()` / `put_user()`

定义于 `include/linux/uaccess.h` / `asm/uaccess.h`

#### 【睡眠】

`put_user()` 和 `get_user()` 用于从用户空间中获取和向用户空间中传出单个值（如 `int`、`char` 或 `long`）。指向用户空间的指针永远不应该直接取消引用：应该使用这些程序复制数据。两者都返回 `-EFAULT` 或 `0`。

`copy_to_user()` 和 `copy_from_user()` 更通用：它们从/向用户空间复制任意数量的数据。

**Warning:** 与 `put_user()` 和 `get_user()` 不同，它们返回未复制的数据量（即 `0` 仍然意味着成功）。

【是的，这个愚蠢的接口真心让我尴尬。火爆的口水仗大概每年都会发生。——Rusty Russell】

这些函数可以隐式睡眠。它不应该在用户上下文之外调用（没有意义）、调用时禁用中断或获得自旋锁。

### `kmalloc()`/`kfree()`

定义于 `include/linux/slab.h`

#### 【可能睡眠：见下】

这些函数用于动态请求指针对齐的内存块，类似用户空间中的 `malloc` 和 `free`，但 `kmalloc()` 需要额外的标志词。重要的值：

**GFP\_KERNEL** 可以睡眠和交换以释放内存。只允许在用户上下文中使用，但这是分配内存最可靠的方法。

**GFP\_ATOMIC** 不会睡眠。较 **GFP\_KERNEL** 更不可靠，但可以从中断上下文调用。你应该有一个很好的内存不足错误处理策略。

**GFP\_DMA** 分配低于 16MB 的 ISA DMA。如果你不知道那是什么，那你就不需要了。非常不可靠。

如果您看到一个从无效上下文警告消息调用的睡眠的函数，那么您可能在没有 **GFP\_ATOMIC** 的情况下从中断上下文调用了一个睡眠的分配函数。你必须立即修复，快点！

如果你要分配至少 `PAGE_SIZE` (`asm/page.h` 或 `asm/page_types.h`) 字节，请考虑使用 `_get_free_pages()` (`include/linux/gfp.h`)。它采用顺序参数 (0 表示页面大小，1 表示双页，2 表示四页……) 和与上述相同的内存优先级标志字。

如果分配的字节数超过一页，可以使用 `vmalloc()`。它将在内核映射中分配虚拟内存。此块在物理内存中不是连续的，但是 MMU（内存管理单元）使它看起来像是为您准备好的连续空间（因此它只是看起来对 CPU 连

续，对外部设备驱动程序则不然）。如果您真的需要为一些奇怪的设备提供大量物理上连续的内存，那么您就会遇到问题：Linux 对此支持很差，因为正在运行的内核中的内存碎片化会使它变得很困难。最好的方法是在引导过程的早期通过 `alloc_bootmem()` 函数分配。

在创建自己的常用对象缓存之前，请考虑使用 `include/linux/slab.h` 中的 slab 缓存。

## current

定义于 `include/asm/current.h`

此全局变量（其实是宏）包含指向当前任务结构（task structure）的指针，因此仅在用户上下文中有效。例如，当进程进行系统调用时，这将指向调用进程的任务结构。在中断上下文中不为空（**not NULL**）。

## mdelay() / udelay()

定义于 `include/asm/delay.h` / `include/linux/delay.h`

`udelay()` 和 `ndelay()` 函数可被用于小暂停。不要对它们使用大的值，因为这样会导致溢出——帮助函数 `mdelay()` 在这里很有用，或者考虑 `msleep()`。

## cpu\_to\_be32() / be32\_to\_cpu() / cpu\_to\_le32() / le32\_to\_cpu()

定义于 `include/asm/byteorder.h`

`cpu_to_be32()` 系列函数（其中“32”可以替换为 64 或 16，“be”可以替换为“le”）是在内核中进行字节序转换的常用方法：它们返回转换后的值。所有的变体也提供反向转换函数：`be32_to_cpu()` 等。

这些函数有两个主要的变体：指针变体，例如 `cpu_to_be32p()`，它获取指向给定类型的指针，并返回转换后的值。另一个变体是“in-situ”系列，例如 `cpu_to_be32s()`，它转换指针引用的值，并返回 `void`。

## local\_irq\_save() / local\_irq\_restore()

定义于 `include/linux/irqflags.h`

这些程序禁用本地 CPU 上的硬中断，并还原它们。它们是可重入的；在其一个 `unsigned long flags` 参数中保存以前的状态。如果您知道中断已启用，那么可直接使用 `local_irq_disable()` 和 `local_irq_enable()`。

### `local_bh_disable()`/`local_bh_enable()`

定义于 `include/linux/bottom_half.h`

这些程序禁用本地 CPU 上的软中断，并还原它们。它们是可重入的；如果之前禁用了软中断，那么在调用这对函数之后仍然会禁用它们。它们阻止软中断和子任务在当前 CPU 上运行。

### `smp_processor_id()`

定义于 `include/linux/smp.h`

`get_cpu()` 禁用抢占（这样您就不会突然移动到另一个 cpu）并返回当前处理器号，介于 0 和 `NR_CPUS` 之间。请注意，CPU 编号不一定是连续的。完成后，使用 `put_cpu()` 再次返回。

如果您知道您不能被另一个任务抢占（即您处于中断上下文中，或已禁用抢占），您可以使用 `smp_processor_id()`。

### `__init`/`__exit`/`__initdata`

定义于 `include/linux/init.h`

引导之后，内核释放一个特殊的部分；用 `__init` 标记的函数和用 `__initdata` 标记的数据结构在引导完成后被丢弃：同样地，模块在初始化后丢弃此内存。`__exit` 用于声明只在退出时需要的函数：如果此文件未编译为模块，则该函数将被删除。请参阅头文件以使用。请注意，使用 `EXPORT_SYMBOL()` 或 `EXPORT_SYMBOL_GPL()` 将标记为 `__init` 的函数导出到模块是没有意义的——这将出问题。

### `__initcall()`/`module_init()`

定义于 `include/linux/init.h` / `include/linux/module.h`

内核的许多部分都作为模块（内核的可动态加载部分）良好服务。使用 `module_init()` 和 `module_exit()` 宏可以简化代码编写，无需 `#ifdef`，即可以作为模块运行或内置在内核中。

`module_init()` 宏定义在模块插入时（如果文件编译为模块）或在引导时调用哪个函数：如果文件未编译为模块，`module_init()` 宏将等效于 `__initcall()`，它通过链接器的魔力确保在引导时调用该函数。

该函数可以返回一个错误值，以导致模块加载失败（不幸的是，如果将模块编译到内核中，则此操作无效）。此函数在启用中断的用户上下文中调用，因此可以睡眠。

## `module_exit()`

定义于 `include/linux/module.h`

这个宏定义了在模块删除时要调用的函数（如果是编译到内核中的文件，则无用武之地）。只有在模块使用计数到零时才会调用它。这个函数也可以睡眠，但不能失败：当它返回时，所有的东西都必须清理干净。

注意，这个宏是可选的：如果它不存在，您的模块将不可移除（除非 `rmmod -f`）。

## `try_module_get()/module_put()`

定义于 `include/linux/module.h`

这些函数会操作模块使用计数，以防止删除（如果另一个模块使用其导出的符号之一，则无法删除模块，参见下文）。在调用模块代码之前，您应该在该模块上调用 `try_module_get()`：若失败，那么该模块将被删除，您应该将其视为不存在。若成功，您就可以安全地进入模块，并在完成后调用模块 `module_put()`。

大多数可注册结构体都有所有者字段，例如在 `struct file_operations` 结构体中，此字段应设置为宏 `THIS_MODULE`。

## 等待队列 `include/linux/wait.h`

### 【睡眠】

等待队列用于等待某程序在条件为真时唤醒另一程序。必须小心使用，以确保没有竞争条件。先声明一个 `wait_queue_head_t`，然后对希望等待该条件的进程声明一个关于它们自己的 `wait_queue_entry_t`，并将其放入队列中。

### 声明

使用 `DECLARE_WAIT_QUEUE_HEAD()` 宏声明一个 `wait_queue_head_t`，或者在初始化代码中使用 `init_waitqueue_head()` 程序。

### 排队

将自己放在等待队列中相当复杂，因为你必须在检查条件之前将自己放入队列中。有一个宏可以来执行此操作：`wait_event_interruptible()` (`include/linux/wait.h`) 第一个参数是等待队列头，第二个参数是计算的表达式；当该表达式为 `true` 时宏返回 0，或者在接收到信号时返回 `-ERESTARTSYS`。`wait_event()` 版本会忽略信号。

### 唤醒排队任务

调用 `wake_up()` (`include/linux/wait.h`)，它将唤醒队列中的所有进程。例外情况：如果有一个进程设置了 `TASK_EXCLUSIVE`，队列的其余部分将不会被唤醒。这个基本函数的其他变体也可以在同一个头文件中使用。

### 原子操作

某些操作在所有平台上都有保证。第一类为操作 `atomic_t` (`include/asm/atomic.h`) 的函数；它包含一个有符号整数（至少 32 位长），您必须使用这些函数来操作或读取 `atomic_t` 变量。`atomic_read()` 和 `atomic_set()` 获取并设置计数器，还有 `atomic_add()`, `atomic_sub()`, `atomic_inc()`, `atomic_dec()` 和 `atomic_dec_and_test()`（如果递减为零，则返回 `true`）。

是的。它在原子变量为零时返回 `true`（即`!=0`）。

请注意，这些函数比普通的算术运算速度慢，因此不应过度使用。

第二类原子操作是在 `unsigned long` (`include/linux/bitops.h`) 上的原子位操作。这些操作通常采用指向位模式（bit pattern）的指针，第 0 位是最低有效位。`set_bit()`, `clear_bit()` 和 `change_bit()` 设置、清除和更改给定位。`test_and_set_bit()`, `test_and_clear_bit()` 和 `test_and_change_bit()` 执行相同的操作，但如果之前设置了位，则返回 `true`；这些对于原子设置标志特别有用。

可以使用大于 `BITS_PER_LONG` 位的位索引调用这些操作。但结果在大端序平台上不太正常，所以最好不要这样做。

### 符号

在内核内部，正常的链接规则仍然适用（即除非用 `static` 关键字将符号声明为文件范围，否则它可以在内核中的任何位置使用）。但是对于模块，会保留一个特殊可导出符号表，该表将入口点限制为内核内部。模块也可以导出符号。

#### `EXPORT_SYMBOL()`

定义于 `include/linux/export.h`

这是导出符号的经典方法：动态加载的模块将能够正常使用符号。

#### `EXPORT_SYMBOL_GPL()`

定义于 `include/linux/export.h`

类似于 `EXPORT_SYMBOL()`，只是 `EXPORT_SYMBOL_GPL()` 导出的符号只能由具有由 `MODULE_LICENSE()` 指定 GPL 兼容许可证的模块看到。这意味着此函数被认为是一个内部实现问题，而不是一个真正的接口。一些维护人员和开发人员在添加一些新的 API 或功能时可能却需要导出 `EXPORT_SYMBOL_GPL()`。

## EXPORT\_SYMBOL\_NS()

定义于 `include/linux/export.h`

这是 `EXPORT_SYMBOL()` 的变体，允许指定符号命名空间。符号名称空间记录于 `Documentation/core-api/symbol-namespaces.rst`。

## EXPORT\_SYMBOL\_NS\_GPL()

定义于 `include/linux/export.h`

这是 `EXPORT_SYMBOL_GPL()` 的变体，允许指定符号命名空间。符号名称空间记录于 `Documentation/core-api/symbol-namespaces.rst`。

## 程序与惯例

### 双向链表 `include/linux/list.h`

内核头文件中曾经有三组链表程序，但这一组是赢家。如果你对一个单链表没有特别迫切的需求，那么这是一个不错的选择。

通常 `list_for_each_entry()` 很有用。

### 返回值惯例

对于在用户上下文中调用的代码，违背 C 语言惯例是很常见的，即返回 0 表示成功，返回负错误值（例如 `-EFAULT`）表示失败。这一开始可能是不直观的，但在内核中相当普遍。

使用 `ERR_PTR()` (`include/linux/err.h`) 将负错误值编码到指针中，然后使用 `IS_ERR()` 和 `PTR_ERR()` 将其再取出：避免为错误值使用单独的指针参数。挺讨厌的，但的确是个好方式。

### 破坏编译

Linus 和其他开发人员有时会更改开发内核中的函数或结构体名称；这样做不仅是为了让每个人都保持警惕，还反映了一个重大的更改（例如，不能再在打开中断的情况下调用，或者执行额外的检查，或者不执行以前捕获的检查）。通常这会附带一个 `linux` 内核邮件列表中相当全面的注释；请搜索存档以查看。简单地对文件进行全局替换通常会让事情变得更糟。

## 初始化结构体成员

初始化结构体的首选方法是使用指定的初始化器，如 ISO C99 所述。例如：

```
static struct block_device_operations opt_fops = {  
    .open          = opt_open,  
    .release       = opt_release,  
    .ioctl         = opt_ioctl,  
    .check_media_change = opt_media_change,  
};
```

这使得很容易查找 (grep)，并且可以清楚地看到设置了哪些结构字段。你应该这样做，因为它看起来很酷。

## GNU 扩展

Linux 内核中明确允许 GNU 扩展。请注意，由于缺乏通用性，一些更复杂的版本并没有得到很好的支持，但以下内容被认为是标准的（有关更多详细信息，请参阅 GCC info 页的“C 扩展”部分——是的，实际上是 info 页，手册页只是 info 中内容的简短摘要）。

- 内联函数
- 语句表达式 (Statement expressions) (即 ({ 和 }) 结构)。
- 声明函数/变量/类型的属性 (`__attribute__`)
- `typeof`
- 零长度数组
- 宏变量
- 空指针运算
- 非常量 (Non-Constant) 初始化程序
- 汇编程序指令 (在 arch/ 和 include/asm/ 之内)
- 字符串函数名 (`__func__`)。
- `__builtin_constant_p()`

在内核中使用 `long long` 时要小心，gcc 为其生成的代码非常糟糕：除法和乘法在 i386 上不能工作，因为内核环境中缺少用于它的 gcc 运行时函数。

## C++

在内核中使用 C++ 通常是个坏主意，因为内核不提供必要的运行时环境，并且不为其测试包含文件。不过这仍然是可能的，但不建议。如果你真的想这么做，至少别用异常处理（exceptions）。

## #if

通常认为，在头文件（或.c 文件顶部）中使用宏来抽象函数比在源代码中使用“if”预处理器语句更干净。

## 把你的东西放进内核里

为了让你的东西更正式、补丁更整洁，还有一些工作要做：

- 搞清楚你在谁的地界儿上干活。查看源文件的顶部、MAINTAINERS 文件以及 CREDITS 文件的最后一部分。你应该和此人协调，确保你没有重新发明轮子，或者尝试一些已经被拒绝的东西。

确保你把你的名字和电子邮件地址放在你创建或修改的任何文件的顶部。当人们发现一个缺陷，或者想要做出修改时，这是他们首先会看的地方。

- 通常你需要一个配置选项来支持你的内核编程。在适当的目录中编辑 Kconfig。配置语言很容易通过剪切和粘贴来使用，在 Documentation/kbuild/kconfig-language.rst 中有完整的文档。

在您对选项的描述中，请确保同时照顾到了专家用户和对此功能一无所知的用户。在此说明任何不兼容和问题。结尾一定要写上“如有疑问，就选 N”（或者是“Y”）；这是针对那些看不懂你在说什么的人的。

- 编辑 Makefile：配置变量在这里导出，因此通常你只需添加一行“obj-\$(CONFIG\_xxx) += xxx.o”。语法记录在 Documentation/kbuild/makefiles.rst 。
- 如果你做了一些有意义的事情，那可以把自己放进 CREDITS，通常不止一个文件（无论如何你的名字都应该在源文件的顶部）。维护人员意味着您希望在对子系统进行更改时得到询问，并了解缺陷；这意味着对某部分代码做出更多承诺。
- 最后，别忘记去阅读 Documentation/process/submitting-patches.rst，也许还有 Documentation/process/submitting-drivers.rst 。

## Kernel 仙女棒

浏览源代码时的一些收藏。请随意添加到此列表。

arch/x86/include/asm/delay.h:

```
#define ndelay(n) (__builtin_constant_p(n) ? \
    ((n) > 20000 ? __bad_ndelay() : __const_udelay((n) * 5ul)) : \
    __ndelay(n))
```

include/linux/fs.h:

```
/*
 * Kernel pointers have redundant information, so we can use a
 * scheme where we can return either an error code or a dentry
 * pointer with the same return value.
 *
 * This should be a per-architecture thing, to allow different
 * error and pointer decisions.
 */
#define ERR_PTR(err)    ((void *)((long)(err)))
#define PTR_ERR(ptr)   ((long)(ptr))
#define IS_ERR(ptr)    ((unsigned long)(ptr) > (unsigned long)(-1000))
```

arch/x86/include/asm/uaccess\_32.h::

```
#define copy_to_user(to,from,n) \
    (__builtin_constant_p(n) ? \
     __constant_copy_to_user((to),(from),(n)) : \
     __generic_copy_to_user((to),(from),(n))) \
```

arch/sparc/kernel/head.S::

```
/*
 * Sun people can't spell worth damn. "compatibility" indeed.
 * At least we *know* we can't spell, and use a spell-checker.
 */
/* Uh, actually Linus it is I who cannot spell. Too much murky
 * Sparc assembly will do this to ya.
 */
C_LABEL(cputypvar):
    .asciz "compatibility"

/* Tested on SS-5, SS-10. Probably someone at Sun applied a spell-checker. */
    .align 4
C_LABEL(cputypvar_sun4m):
    .asciz "compatible"
```

arch/sparc/lib/checksum.S::

```
/* Sun, you just can't beat me, you just can't. Stop trying,
 * give up. I'm serious, I am going to kick the living shit
 * out of you, game over, lights out.
 */
```

## 致谢

感谢 Andi Kleen 提出点子，回答我的问题，纠正我的错误，充实内容等帮助。感谢 Philipp Rumpf 做了许多拼写和清晰度修复，以及一些优秀的不明显的点。感谢 Werner Almesberger 对 disable\_irq() 做了一个很好的总结，Jes Sorensen 和 Andrea Arcangeli 补充了一些注意事项。感谢 Michael Elizabeth Chastain 检查并补充了配置部分。感谢 Telsa Gwynne 教我 DocBook。

## TODO

- locking

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:<alexs@kernel.org>)。

---

## Original Documentation/maintainer/index.rst

### \* 内核维护者手册

本文档本是内核维护者手册的首页。本手册还需要大量完善！请自由提出（和编写）本手册的补充内容。译注：指英文原版

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:<alexs@kernel.org>)。

---

## Original Documentation/maintainer/configure-git.rst

译者 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

### Git 配置

本章讲述了维护者级别的 git 配置。

Documentation/maintainer/pull-requests.rst 中使用的标记分支应使用开发人员的 GPG 公钥进行签名。可以通过将 -u 标志传递给 git tag 来创建签名标记。但是，由于 通常对同一项目使用同一个密钥，因此可以设置：

```
git config user.signingkey "keyname"
```

或者手动编辑你的 .git/config 或 ~/.gitconfig 文件：

```
[user]
  name = Jane Developer
  email = jd@domain.org
  signingkey = jd@domain.org
```

你可能需要告诉 git 去使用 gpg2：

```
[gpg]
  program = /path/to/gpg2
```

你可能也需要告诉 gpg 去使用哪个 tty （添加到你的 shell rc 文件中）：

```
export GPG_TTY=$(tty)
```

### 创建链接到 [lore.kernel.org](https://lore.kernel.org) 的提交

<http://lore.kernel.org> 网站是所有涉及或影响内核开发的邮件列表的总存档。在这里存储补丁存档是推荐的做法，当维护人员将补丁应用到子系统树时，最好提供一个指向 lore 存档链接的标签，以便浏览提交历史的人可以找到某个更改背后的相关讨论和基本原理。链接标签如下所示：

Link: <https://lore.kernel.org/r/<message-id>>

通过在 git 中添加以下钩子，可以将此配置为在发布 git am 时自动执行：

```
$ git config am.messageid true
$ cat > .git/hooks/applypatch-msg <<'EOF'
#!/bin/sh
. git-sh-setup
perl -pi -e 's|^Message-Id:\s*([&gt;]+)&gt;?|Link: https://lore.kernel.org/r/$1|g;' "$1"
test -x "$GIT_DIR/hooks/commit-msg" &amp;&amp;
  exec "$GIT_DIR/hooks/commit-msg" ${1+"$@"}
:
EOF
$ chmod a+x .git/hooks/applypatch-msg</pre
```

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/maintainer/rebasing-and-merging.rst

译者 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

## 变基与合并

一般来说，维护子系统需要熟悉 Git 源代码管理系统。Git 是一个功能强大的工具，有很多功能；就像这类工具常出现的情况一样，使用这些功能的方法有对有错。本文档特别介绍了变基与合并的用法。维护者经常在错误使用这些工具时遇到麻烦，但避免问题实际上并不那么困难。

总的来说，需要注意的一点是：与许多其他项目不同，内核社区并不害怕在其开发历史中看到合并提交。事实上，考虑到该项目的规模，避免合并几乎是不可能的。维护者会在希望避免合并时遇到一些问题，而过于频繁的合并也会带来另一些问题。

## 变基

“变基（Rebase）”是更改存储库中一系列提交的历史记录的过程。有两种不同型的操作都被称为变基，因为这两种操作都使用 `git rebase` 命令，但它们之间存在显著差异：

- 更改一系列补丁的父提交（起始提交）。例如，变基操作可以将基于上一内核版本的一个补丁集重建到当前版本上。在下面的讨论中，我们将此操作称为“变根”。
- 通过修复（或删除）损坏的提交、添加补丁、添加标记以更改一系列补丁的历史，来提交变更日志或更改已应用提交的顺序。在下文中，这种类型的操作称为“历史修改”

术语“变基”将用于指代上述两种操作。如果使用得当，变基可以产生更清晰、更整洁的开发历史；如果使用不当，它可能会模糊历史并引入错误。

以下一些经验法则可以帮助开发者避免最糟糕的变基风险：

- 已经发布到你私人系统之外世界的历史通常不应更改。其他人可能会拉取你的树的副本，然后基于它进行工作；修改你的树会给他们带来麻烦。如果工作需要变基，这通常是表明它还没有准备好提交到公共存储库的信号。

但是，总有例外。有些树（linux-next 是一个典型的例子）由于它们的需要经常变基，开发人员知道不要基于它们来工作。开发人员有时会公开一个不稳定的分支，供其他人或自动测试服务进行测试。如果您确实以这种方式公开了一个可能不稳定的分支，请确保潜在使用者知道不要基于它来工作。

- 不要在包含由他人创建的历史的分支上变基。如果你从别的开发者的仓库拉取了变更，那你现在就成了他们历史记录的保管人。你不应该改变它，除了少数例外情况。例如树中有问题的提交必须显式恢复（即通过另一个补丁修复），而不是通过修改历史而消失。
- 没有合理理由，不要对树变根。仅为了切换到更新的基或避免与上游储存库的合并通常不是合理理由。
- 如果你必须对储存库进行变根，请不要随机选取一个提交作为新基。在发布节点之间内核通常处于一个相对不稳定的状态；基于其中某点进行开发会增加遇到意外错误的几率。当一系列补丁必须移动到新基时，请选择移动到一个稳定节点（例如-rc 版本节点）。
- 请知悉对补丁系列进行变根（或做明显的历史修改）会改变它们的开发环境，且很可能使做过的大部分测试失效。一般来说，变基后的补丁系列应当像新代码一样对待，并重新测试。

合并窗口麻烦的一个常见原因是，Linus 收到了一个明显在拉取请求发送之前不久才变根（通常是变根到随机的提交上）的补丁系列。这样一个系列被充分测试的可能性相对较低，拉取请求被接受的几率也同样较低。

相反，如果变基仅限于私有树、提交基于一个通用的起点、且经过充分测试，则引起麻烦的可能性就很低。

## 合并

内核开发过程中，合并是一个很常见的操作；5.1 版本开发周期中有超过 1126 个合并——差不多占了整体的 9%。内核开发工作积累在 100 多个不同的子系统树中，每个子系统树都可能包含多个主题分支；每个分支通常独立于其他分支进行开发。因此在任何给定分支进入上游储存库之前，至少需要一次合并。

许多项目要求拉取请求中的分支基于当前主干，这样历史记录中就不会出现合并提交。内核并不是这样；任何为了避免合并而重新对分支变基都很可能导致麻烦。

子系统维护人员发现他们必须进行两种类型的合并：从较低层级的子系统树和从其他子系统树（同级树或主线）进行合并。这两种情况下要遵循的最佳实践是不同的。

### 合并较低层级树

较大的子系统往往有多个级别的维护人员，较低级别的维护人员向较高级别发送拉取请求。合并这样的请求执行几乎肯定会生成一个合并提交；这也是应该的。实际上，子系统维护人员可能希望在极少数快进合并情况下使用 `--no-ff` 标志来强制添加合并提交，以便记录合并的原因。**任何类型的合并的变更日志必须说明为什么合并**。对于较低级别的树，“为什么”通常是对该取所带来的变化的总结。

各级维护人员都应在他们的拉取请求上使用经签名的标签，上游维护人员应在拉取分支时验证标签。不这样做会威胁整个开发过程的安全。

根据上面列出的规则，一旦您将其他人的历史记录合并到树中，您就不得对该分支进行变基，即使您能够这样做。

## 合并同级树或上游树

虽然来自下游的合并是常见且不起眼的，但当需要将一个分支推向上游时，其中来自其他树的合并往往是一个危险信号。这种合并需要仔细考虑并加以充分证明，否则后续的拉取请求很可能被拒绝。

想要将主分支合并到存储库中是很自然的；这种类型的合并通常被称为“反向合并”。反向合并有助于确保与并行的开发没有冲突，并且通常会给人一种温暖、舒服的感觉，即处于最新。但这种诱惑几乎总是应该避免的。

为什么呢？反向合并将搅乱你自己分支的开发历史。它们会大大增加你遇到来自社区其他地方的错误的机会，且使你很难确保你所管理的工作稳定并准备好合入上游。频繁的合并还可以掩盖树中开发过程中的问题；它们会隐藏与其他树的交互，而这些交互不应该（经常）发生在管理良好的分支中。

也就是说，偶尔需要进行反向合并；当这种情况发生时，一定要在提交信息中记录 **为什么**。同样，在一个众所周知的稳定点进行合并，而不是随机提交。即使这样，你也不应该反向合并一棵比你的直接上游树更高层级的树；如果确实需要更高级别的反向合并，应首先在上游树进行。

导致合并相关问题最常见的原因之一是：在发送拉取请求之前维护者合并上游以解决合并冲突。同样，这种诱惑很容易理解，但绝对应该避免。对于最终拉取请求来说尤其如此：Linus 坚信他更愿意看到合并冲突，而不是不必要的反向合并。看到冲突可以让他了解潜在的问题所在。他做过很多合并（在 5.1 版本开发周期中是 382 次），而且在解决冲突方面也很在行——通常比参与的开发人员要强。

那么，当他们的子系统分支和主线之间发生冲突时，维护人员应该怎么做呢？最重要的一步是在拉取请求中提示 Linus 会发生冲突；如果啥都没说则表明您的分支可以正常合入。对于特别困难的冲突，创建并推送一个独立分支来展示你将如何解决问题。在拉取请求中提到该分支，但是请求本身应该针对未合并的分支。

即使不存在已知冲突，在发送拉取请求之前进行合并测试也是个好主意。它可能会提醒您一些在 linux-next 树中没有发现的问题，并帮助您准确地理解您正在要求上游做什么。

合并上游树或另一个子系统树的另一个原因是解决依赖关系。这些依赖性问题有时确实会发生，而且有时与另一棵树交叉合并是解决这些问题的最佳方法；同样，在这种情况下，合并提交应该解释为什么要进行合并。花点时间把它做好；会有人阅读这些变更日志。

然而依赖性问题通常表明需要改变方法。合并另一个子系统树以解决依赖性风险会带来其他缺陷，几乎永远不应这样做。如果该子系统树无法被合到上游，那么它的任何问题也都会阻碍你的树合并。更可取的选择包括与维护人员达成一致意见，在其中一个树中同时进行两组更改；或者创建一个主题分支专门处理可以合并到两个树中的先决条件提交。如果依赖关系与主要的基础结构更改相关，正确的解决方案可能是将依赖提交保留一个开发周期，以便这些更改有时间在主线上稳定。

## 最后

在开发周期的开头合并主线是比较常见的，可以获取树中其他地方的更改和修复。同样，这样的合并应该选择一个众所周知的发布点，而不是一些随机点。如果在合并窗口期间上游分支已完全清空到主线中，则可以使用以下命令向前拉取它：

```
git merge v5.2-rc1^0
```

“`^0`”使 Git 执行快进合并（在这种情况下这应该可以），从而避免多余的虚假合并提交。

上面列出的就是指导方针了。总是会有一些情况需要不同的解决方案，这些指导原则不应阻止开发人员在需要时做正确的事情。但是，我们应该时刻考虑是否真的出现了这样的需求，并准备好解释为什么需要做一些不寻常的事情。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

### Original Documentation/maintainer/pull-requests.rst

译者 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

## 如何创建拉取请求

本章描述维护人员如何创建并向其他维护人员提交拉取请求。这对将更改从一个维护者树转移到另一个维护者树非常有用。

本文档由 Tobin C. Harding（当时他尚不是一名经验丰富的维护人员）编写，内容主要来自 Greg Kroah Hartman 和 Linus Torvalds 在 LKML 上的评论。Jonathan Corbet 和 Mauro Carvalho Chehab 提出了一些建议和修改。错误不可避免，如有问题，请找 Tobin C. Harding <[me@tobin.cc](mailto:me@tobin.cc)>。

原始邮件线程：

<https://lore.kernel.org/r/20171114110500.GA21175@kroah.com>

## 创建分支

首先，您需要将希望包含拉取请求里的所有更改都放在单独分支中。通常您将基于某开发人员树的一个分支，一般是打算向其发送拉取请求的开发人员。

为了创建拉取请求，您必须首先标记刚刚创建的分支。建议您选择一个有意义的标记名，以即使过了一段时间您和他人仍能理解的方式。在名称中包含源子系统和目标内核版本的指示也是一个好的做法。

Greg 提供了以下内容。对于一个含有 drivers/char 中混杂事项、将应用于 4.15-rc1 内核的拉取请求，可以命名为 char-misc-4.15-rc1。如果要在 char-misc-next 分支上打上此标记，您可以使用以下命令：

```
git tag -s char-misc-4.15-rc1 char-misc-next
```

这将在 char-misc-next 分支的最后一个提交上创建一个名为 char-misc-4.15-rc1 的标记，并用您的 gpg 密钥签名（参见 Documentation/maintainer/configure-git.rst）。

Linus 只接受基于签名过的标记的拉取请求。其他维护者可能会有所不同。

当您运行上述命令时 git 会打开编辑器要求你描述一下这个标记。在本例中您需要描述拉取请求，所以请概述一下包含的内容，为什么要合并，是否完成任何测试。所有这些信息都将留在标记中，然后在维护者合并拉取请求时保留在合并提交中。所以把它写好，它将永远留在内核中。

正如 Linus 所说：

不管怎么样，至少对我来说，重要的是 \* 信息 \*。我需要知道我在拉取什么、为什么我要拉取。我也希望将此消息用于合并消息，因此它不仅应该对我有意义，也应该可以成为一个有意义的历史记录。

注意，如果拉取请求有一些不寻常的地方，请详细说明。如果你修改了并非由你维护的文件，请解释 \*\* 为什么 \*\*。我总会在差异中看到的，如果你不提的话，我只会觉得分外可疑。当你在合并窗口后给我发新东西的时候，(甚至是比较重大的错误修复)，不仅需要解释做了什么、为什么这么做，还请解释一下 \*\* 时间问题 \*\*。为什么错过了合并窗口……

我会看你写在拉取请求邮件和签名标记里面的内容，所以根据你的工作流，你可以在签名标记里面描述作品内容（也会自动放进拉取请求邮件），也可以只在标记里面放个占位符，稍后在你实际发给我拉取请求时描述作品内容。

是的，我会编辑这些消息。部分因为我需要做一些琐碎的格式调整（整体缩进、括号等），也因为此消息可能对我有意义（描述了冲突或一些个人问题）而对合并提交信息上下文没啥意义，因此我需要尽力让它有意义起来。我也会修复一些拼写和语法错误，特别是非母语者（母语者也是;^）。但我也会删掉或增加一些内容。

Linus

Greg 给出了一个拉取请求的例子：

Char/Misc patches for 4.15-rc1

Here is the big char/misc patch set for the 4.15-rc1 merge window. Contained in here is the normal set of new functions added to all of these crazy drivers, as well as the following brand new subsystems:

- time\_travel\_controller: Finally a set of drivers for the latest time travel bus architecture that provides i/o to the CPU before it asked for it, allowing uninterrupted processing
- relativity\_shifters: due to the affect that the time\_travel\_controllers have on the overall system, there was a need for a new set of relativity shifter drivers to accommodate the newly formed black holes that would threaten to suck CPUs into them. This subsystem handles this in a way to successfully neutralize the problems. There is a Kconfig option to force these to be enabled when needed, so problems should not occur.

All of these patches have been successfully tested in the latest linux-next releases, and the original problems that it found have all been resolved (apologies to anyone living near Canberra for the lack of the Kconfig options in the earlier versions of the linux-next tree creations.)

Signed-off-by: Your-name-here <your\_email@domain>

此标记消息格式就像一个 git 提交。顶部有一行“总结标题”，一定要在下面 sign-off。

现在您已经有了一个本地签名标记，您需要将它推送到可以被拉取的位置：

```
git push origin char-misc-4.15-rc1
```

### 创建拉取请求

最后要做的是创建拉取请求消息。可以使用 `git request-pull` 命令让 git 为你做这件事，但它需要确定你想拉取什么，以及拉取针对的基础（显示正确的拉取更改和变更状态）。以下命令将生成一个拉取请求：

```
git request-pull master git://git.kernel.org/pub/scm/linux/kernel/git/gregkh/char-misc.git/  
→char-misc-4.15-rc1
```

引用 Greg 的话：

此命令要求 git 比较从“char-misc-4.15-rc1”标记位置到“master”分支头（上述例子中指向了我从 Linus 的树分叉的地方，通常是-rc 发布）的差异，并去使用 `git://` 协议拉取。如果你希望使用 `https://` 协议，也可以用在这里（但是请注意，部分人由于防火墙问题没法用 `https` 协议拉取）。

如果 `char-misc-4.15-rc1` 标记没有出现在我要求拉取的仓库中，git 会提醒它不在那里，所以记得推送到公开地方。

“`git request-pull`”会包含 git 树的地址和需要拉取的特定标记，以及标记描述全文（详尽描述标记）。同时它也会创建此拉取请求的差异状态和单个提交的缩短日志。

Linus 回复说他倾向于 `git://` 协议。其他维护者可能有不同的偏好。另外，请注意如果你创建的拉取请求没有签名标记，`https://` 可能是更好的选择。完整的讨论请看原邮件。

## 提交拉取请求

拉取请求的提交方式与普通补丁相同。向维护人员发送内联电子邮件并抄送 LKML 以及任何必要特定子系统的列表。对 Linus 的拉取请求通常有如下主题行：

```
[GIT PULL] <subsystem> changes for v4.15-rc1
```

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

**Original** Documentation/maintainer/maintainer-entry-profile.rst

译者 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

## 维护者条目概要

维护人员条目概要补充了顶层过程文档（提交补丁，提交驱动程序……），增加了子系统/设备驱动程序本地习惯以及有关补丁提交生命周期的相关内容。贡献者使用此文档来调整他们的期望和避免常见错误；维护人员可以使用这些信息超越子系统层面查看是否有机会汇聚到通用实践中。

## 总览

提供了子系统如何操作的介绍。MAINTAINERS 文件告诉了贡献者应发送某文件的补丁到哪，但它没有传达其他子系统的本地基础设施和机制以协助开发。

请考虑以下问题：

- 当补丁被本地树接纳或合并到上游时是否有通知？
- 子系统是否使用 patchwork 实例？Patchwork 状态变更是否有通知？
- 是否有任何机器人或 CI 基础设施监视列表，或子系统是否使用自动测试反馈以便把控接纳补丁？
- 被拉入-next 的 Git 分支是哪个？
- 贡献者应针对哪个分支提交？
- 是否链接到其他维护者条目概要？例如一个设备驱动可能指向其父子系统的条目。这使得贡献者意识到某维护者可能对提交链中其他维护者负有的义务。

### 提交检查单补遗

列出强制性和咨询性标准，超出通用标准“提交检查表，以便维护者检查一个补丁是否足够健康。例如：“通过 checkpatch.pl，没有错误、没有警告。通过单元测试详见某处”。

提交检查单补遗还可以包括有关硬件规格状态的详细信息。例如，子系统接受补丁之前是否需要考虑在某个修订版上发布的规范。

### 开发周期的关键日期

提交者常常会误以为补丁可以在合并窗口关闭之前的任何时间发送，且下一个-rc1 时仍可以。事实上，大多数补丁都需要在下一个合并窗口打开之前提前进入 linux-next 中。向提交者澄清关键日期（以-rc 发布周为标志）以明确什么时候补丁会被考虑合并以及何时需要等待下一个-rc。

至少需要讲明：

- 最后一个可以提交新功能的-rc：针对下一个合并窗口的新功能提交应该在此点之前首次发布以供考虑。在此时间点之后提交的补丁应该明确他们的目标为下下个合并窗口，或者给出应加快进度被接受的充足理由。通常新特性贡献者的提交应出现在-rc5 之前。
- 最后合并-rc：合并决策的最后期限。向贡献者指出尚未接受的补丁集需要等待下下个合并窗口。当然，维护者没有义务接受所有给定的补丁集，但是如果审阅在此时间点尚未结束，那么希望贡献者应该等待并在下一个合并窗口重新提交。

可选项：

- 开发基线分支的首个-rc，列在概述部分，视为已为新提交做好准备。

### 审阅节奏

贡献者最担心的问题之一是：补丁集已发布却未收到反馈，应在多久后发送提醒。除了指定在重新提交之前要等待多长时间，还可以指示更新的首选样式；例如，重新发送整个系列，或私下发送提醒邮件。本节也可以列出本区域的代码审阅方式，以及获取不能直接从维护者那里得到的反馈的方法。

### 现有概要

这里列出了现有的维护人员条目概要；我们可能会想要在不久的将来做一些不同的事情。

## LIBNVDIMM Maintainer Entry Profile

### Overview

The libnvdimm subsystem manages persistent memory across multiple architectures. The mailing list is tracked by patchwork here: <https://patchwork.kernel.org/project/linux-nvdimm/list/> ...and that instance is configured to give feedback to submitters on patch acceptance and upstream merge. Patches are merged to either the ‘libnvdimm-fixes’ or ‘libnvdimm-for-next’ branch. Those branches are available here: <https://git.kernel.org/pub/scm/linux/kernel/git/nvdimm/nvdimm.git/>

In general patches can be submitted against the latest -rc; however, if the incoming code change is dependent on other pending changes then the patch should be based on the libnvdimm-for-next branch. However, since persistent memory sits at the intersection of storage and memory there are cases where patches are more suitable to be merged through a Filesystem or the Memory Management tree. When in doubt copy the nvdimm list and the maintainers will help route.

Submissions will be exposed to the kbuild robot for compile regression testing. It helps to get a success notification from that infrastructure before submitting, but it is not required.

### Submit Checklist Addendum

There are unit tests for the subsystem via the ndctl utility: <https://github.com/pmem/ndctl> Those tests need to be passed before the patches go upstream, but not necessarily before initial posting. Contact the list if you need help getting the test environment set up.

### ACPI Device Specific Methods (\_DSM)

Before patches enabling a new \_DSM family will be considered, it must be assigned a format-interface-code from the NVDIMM Sub-team of the ACPI Specification Working Group. In general, the stance of the subsystem is to push back on the proliferation of NVDIMM command sets, so do strongly consider implementing support for an existing command set. See drivers/acpi/nfit/nfit.h for the set of supported command sets.

### Key Cycle Dates

New submissions can be sent at any time, but if they intend to hit the next merge window they should be sent before -rc4, and ideally stabilized in the libnvdimm-for-next branch by -rc6. Of course if a patch set requires more than 2 weeks of review, -rc4 is already too late and some patches may require multiple development cycles to review.

### Review Cadence

In general, please wait up to one week before pinging for feedback. A private mail reminder is preferred. Alternatively ask for other developers that have Reviewed-by tags for libnvdimm changes to take a look and offer their opinion.

## arch/riscv maintenance guidelines for developers

### Overview

The RISC-V instruction set architecture is developed in the open: in-progress drafts are available for all to review and to experiment with implementations. New module or extension drafts can change during the development process - sometimes in ways that are incompatible with previous drafts. This flexibility can present a challenge for RISC-V Linux maintenance. Linux maintainers disapprove of churn, and the Linux development process prefers well-reviewed and tested code over experimental code. We wish to extend these same principles to the RISC-V-related code that will be accepted for inclusion in the kernel.

### Submit Checklist Addendum

We'll only accept patches for new modules or extensions if the specifications for those modules or extensions are listed as being "Frozen" or "Ratified" by the RISC-V Foundation. (Developers may, of course, maintain their own Linux kernel trees that contain code for any draft extensions that they wish.)

Additionally, the RISC-V specification allows implementors to create their own custom extensions. These custom extensions aren't required to go through any review or ratification process by the RISC-V Foundation. To avoid the maintenance complexity and potential performance impact of adding kernel code for implementor-specific RISC-V extensions, we'll only accept patches for extensions that have been officially frozen or ratified by the RISC-V Foundation. (Implementors, may, of course, maintain their own Linux kernel trees containing code for any custom extensions that they wish.)

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

### Original Documentation/maintainer/modifying-patches.rst

译者 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

## 修改补丁

如果你是子系统或者分支的维护者，由于代码在你的和提交者的树中并不完全相同，有时你需要稍微修改一下收到的补丁以合并它们。

如果你严格遵守开发者来源证书的规则 (c)，你应该要求提交者重做，但这完全是会适得其反的时间、精力浪费。规则 (b) 允许你调整代码，但这样修改提交者的代码并让他背书你的错误是非常不礼貌的。为解决此问题，建议在你之前最后一个 Signed-off-by 标签和你的之间添加一行，以指示更改的性质。这没有强制性要求，最好在描述前面加上你的邮件和/或姓名，用方括号括住整行，以明显指出你对最后一刻的更改负责。例如：

```
Signed-off-by: Random J Developer <random@developer.example.org>
[lucky@maintainer.example.org: struct foo moved from foo.c to foo.h]
Signed-off-by: Lucky K Maintainer <lucky@maintainer.example.org>
```

如果您维护着一个稳定的分支，并希望同时明确贡献、跟踪更改、合并修复，并保护提交者免受责难，这种做法尤其有用。请注意，在任何情况下都不得更改作者的身份 (From 头)，因为它会在变更日志中显示。

向后移植 (back-port) 人员特别要注意：为了便于跟踪，请在提交消息的顶部（即主题行之后）插入补丁的来源，这是一种常见而有用的做法。例如，我们可以在 3.x 稳定版本中看到以下内容：

```
Date: Tue Oct 7 07:26:38 2014 -0400

libata: Un-break ATA blacklist

commit 1c40279960bcd7d52dbdf1d466b20d24b99176c8 upstream.
```

下面是一个旧的内核在某补丁被向后移植后会出现的：

```
Date: Tue May 13 22:12:27 2008 +0200

wireless, airo: waitbusy() won't delay

[backport of 2.6 commit b7acbd1f277c1eb23f344f899cfa4cd0bf36a]
```

不管什么格式，这些信息都为人们跟踪你的树，以及试图解决你树中的错误的人提供了有价值的帮助。

TODOList:

- trace/index
- fault-injection/index
- livepatch/index
- rust/index

## \* 内核 API 文档

以下手册从内核开发人员的角度详细介绍了特定的内核子系统是如何工作的。这里的大部分信息都是直接从内核源代码获取的，并根据需要添加补充材料（或者至少是在我们设法添加的时候——可能不是所有的都是有需要的）。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

**Original** Documentation/core-api/index.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## \* 核心 API 文档

这是核心内核 API 手册的首页。非常感谢为本手册转换（和编写！）的文档！

### 核心实用程序

本节包含通用的和“核心中的核心”文档。第一部分是 docbook 时期遗留下来的大量 kerneldoc 信息；有朝一日，若有人有动力的话，应当把它们拆分出来。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题, 请联系该文件的译者, 或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/core-api/kernel-api.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## Linux 内核 API

### 列表管理函数

该 API 在以下内核代码中:

include/linux/list.h

### 基本的 C 库函数

在编写驱动程序时, 一般不能使用 C 库中的例程。部分函数通常很有用, 它们在下面被列出。这些函数的行为可能会与 ANSI 定义的略有不同, 这些偏差会在文中注明。

### 字符串转换

该 API 在以下内核代码中:

lib/vsprintf.c

include/linux/kernel.h

include/linux/kernel.h

lib/kstrtox.c

lib/string\_helpers.c

### 字符串处理

该 API 在以下内核代码中:

lib/string.c

include/linux/string.h

mm/util.c

### 基本的内核库函数

Linux 内核提供了很多实用的基本函数。

### 位运算

该 API 在以下内核代码中:

```
include/asm-generic/bitops/instrumented-atomic.h  
include/asm-generic/bitops/instrumented-non-atomic.h  
include/asm-generic/bitops/instrumented-lock.h
```

### 位图运算

该 API 在以下内核代码中:

```
lib(bitmap.c  
include/linux(bitmap.h  
include/linux(bitmap.h  
include/linux(bitmap.h  
lib(bitmap.c  
lib(bitmap.c  
include/linux(bitmap.h
```

### 命令行解析

该 API 在以下内核代码中:

```
lib/cmdline.c
```

### 排序

该 API 在以下内核代码中:

```
lib/sort.c  
lib/list_sort.c
```

## 文本检索

该 API 在以下内核代码中:

lib/textsearch.c

lib/textsearch.c

include/linux/textsearch.h

## Linux 中的 CRC 和数学函数

### CRC 函数

译注: *CRC*, *Cyclic Redundancy Check*, 循环冗余校验

该 API 在以下内核代码中:

lib/crc4.c

lib/crc7.c

lib/crc8.c

lib/crc16.c

lib/crc32.c

lib/crc-ccitt.c

lib/crc-itu-t.c

## 基数为 2 的对数和幂函数

该 API 在以下内核代码中:

include/linux/log2.h

## 整数幂函数

该 API 在以下内核代码中:

lib/math/int\_pow.c

lib/math/int\_sqrt.c

### 除法函数

该 API 在以下内核代码中:

include/asm-generic/div64.h

include/linux/math64.h

lib/math/div64.c

lib/math/gcd.c

### UUID/GUID

该 API 在以下内核代码中:

lib/uuid.c

### 内核 IPC 设备

#### IPC 实用程序

该 API 在以下内核代码中:

ipc/util.c

### FIFO 缓冲区

#### kfifo 接口

该 API 在以下内核代码中:

include/linux/kfifo.h

### 转发接口支持

转发接口支持旨在为工具和设备提供一种有效的机制，将大量数据从内核空间转发到用户空间。

## 转发接口

该 API 在以下内核代码中:

kernel/relay.c

kernel/relay.c

## 模块支持

### 模块加载

该 API 在以下内核代码中:

kernel/kmod.c

### 模块接口支持

更多信息请参阅 kernel/module/目录下的文件。

## 硬件接口

该 API 在以下内核代码中:

kernel/dma.c

## 资源管理

该 API 在以下内核代码中:

kernel/resource.c

kernel/resource.c

## MTRR 处理

该 API 在以下内核代码中:

arch/x86/kernel/cpu/mtrr/mtrr.c

## **安全框架**

该 API 在以下内核代码中:

security/security.c

security/inode.c

## **审计接口**

该 API 在以下内核代码中:

kernel/audit.c

kernel/auditsc.c

kernel/auditfilter.c

## **核算框架**

该 API 在以下内核代码中:

kernel/acct.c

## **块设备**

该 API 在以下内核代码中:

block/blk-core.c

block/blk-core.c

block/blk-map.c

block/blk-sysfs.c

block/blk-settings.c

block/blk-flush.c

block/blk-lib.c

block/blk-integrity.c

kernel/trace/blktrace.c

block/genhd.c

block/genhd.c

## 字符设备

该 API 在以下内核代码中:

fs/char\_dev.c

## 时钟框架

时钟框架定义了编程接口，以支持系统时钟树的软件管理。该框架广泛用于系统级芯片（SOC）平台，以支持电源管理和各种可能需要自定义时钟速率的设备。请注意，这些“时钟”与计时或实时时钟（RTC）无关，它们都有单独的框架。这些:c:type: *struct clk <clk>* 实例可用于管理各种时钟信号，例如一个 96 MHz 的时钟信号，该信号可被用于总线或外设的数据交换，或以其他方式触发系统硬件中的同步状态机转换。

通过明确的软件时钟门控来支持电源管理：未使用的时钟被禁用，因此系统不会因为改变不在使用中的晶体管的状态而浪费电源。在某些系统中，这可能是由硬件时钟门控支持的，其中时钟被门控而不是软件中被禁用。芯片的部分，在供电但没有时钟的情况下，可能会保留其最后的状态。这种低功耗状态通常被称为 \* 保留模式 \*。这种模式仍然会产生漏电流，特别是在电路几何结构较细的情况下，但对于 CMOS 电路来说，电能主要是随着时钟翻转而被消耗的。

电源感知驱动程序只有在其管理的设备处于活动使用状态时才会启用时钟。此外，系统睡眠状态通常根据哪些时钟域处于活动状态而有所不同：“待机”状态可能允许从多个活动域中唤醒，而“mem”（暂停到 RAM）状态可能需要更全面地关闭来自高速 PLL 和振荡器的时钟，从而限制了可能的唤醒事件源的数量。驱动器的暂停方法可能需要注意目标睡眠状态的系统特定时钟约束。

一些平台支持可编程时钟发生器。这些可以被各种外部芯片使用，如其他 CPU、多媒体编解码器以及对接口时钟有严格要求的设备。

该 API 在以下内核代码中:

include/linux/clk.h

## 同步原语

### 读-复制-更新（RCU）

该 API 在以下内核代码中:

include/linux/rcupdate.h

kernel/rcu/tree.c

kernel/rcu/tree\_exp.h

kernel/rcu/update.c

include/linux/srcu.h

kernel/rcu/srcutree.c  
include/linux/rculist\_bl.h  
include/linux/rculist.h  
include/linux/rculist\_nulls.h  
include/linux/rcu\_sync.h  
kernel/rcu/sync.c

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/core-api/printk-basics.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

### 使用 printk 记录消息

printk() 是 Linux 内核中最广为人知的函数之一。它是我们打印消息的标准工具，通常也是追踪和调试的最基本方法。如果你熟悉 printf(3)，你就能够知道 printk() 是基于它的，尽管它在功能上有一些不同之处：

- printk() 消息可以指定日志级别。
- 格式字符串虽然与 C99 基本兼容，但并不遵循完全相同的规范。它有一些扩展和一些限制（没有 %n 或浮点转换指定符）。参见:ref: 如何正确地获得 printk 格式指定符 <*printk-specifiers*> 。

所有的 printk() 消息都会被打印到内核日志缓冲区，这是一个通过/dev/kmsg 输出到用户空间的环形缓冲区。读取它的通常方法是使用 dmesg 。

printk() 的用法通常是这样的：

```
printk(KERN_INFO "Message: %s\n", arg);
```

其中 KERN\_INFO 是日志级别（注意，它与格式字符串连在一起，日志级别不是一个单独的参数）。可用的日志级别是：

名称	字符串	别名函数
KERN_EMERG	“0”	pr_emerg()
KERN_ALERT	“1”	pr_alert()
KERN_CRIT	“2”	pr_crit()
KERN_ERR	“3”	pr_err()
KERN_WARNING	“4”	pr_warn()
KERN_NOTICE	“5”	pr_notice()
KERN_INFO	“6”	pr_info()
KERN_DEBUG	“7”	pr_debug() and pr-devel() 若定义了 DEBUG
KERN_DEFAULT	“”	
KERN_CONT	“c”	pr_cont()

日志级别指定了一条消息的重要性。内核根据日志级别和当前 *console\_loglevel* (一个内核变量) 决定是否立即显示消息 (将其打印到当前控制台)。如果消息的优先级比 *console\_loglevel* 高 (日志级别值较低), 消息将被打印到控制台。

如果省略了日志级别, 则以 KERN\_DEFAULT 级别打印消息。

你可以用以下方法检查当前的 *console\_loglevel*

```
$ cat /proc/sys/kernel/printk
4      4      1      7
```

结果显示了 *current*, *default*, *minimum* 和 *boot-time-default* 日志级别

要改变当前的 *console\_loglevel*, 只需在 /proc/sys/kernel/printk 中写入所需的级别。例如, 要打印所有的消息到控制台上:

```
# echo 8 > /proc/sys/kernel/printk
```

另一种方式, 使用 dmesg:

```
# dmesg -n 5
```

设置 *console\_loglevel* 打印 KERN\_WARNING (4) 或更严重的消息到控制台。更多消息参见 dmesg(1)。

作为 printk() 的替代方案, 你可以使用 pr\_\*() 别名来记录日志。这个系列的宏在宏名中嵌入了日志级别。例如:

```
pr_info("Info message no. %d\n", msg_num);
```

打印 KERN\_INFO 消息。

除了比等效的 printk() 调用更简洁之外, 它们还可以通过 pr\_fmt() 宏为格式字符串使用一个通用的定义。例如, 在源文件的顶部 (在任何 #include 指令之前) 定义这样的内容。:

```
#define pr_fmt(fmt) "%s:%s: " fmt, KBUILD_MODNAME, __func__
```

会在该文件中的每一条 `pr_*`() 消息前加上发起该消息的模块和函数名称。

为了调试，还有两个有条件编译的宏：`pr_debug()` 和 `pr_devel()`，除非定义了 `DEBUG` (或者在 `pr_debug()` 的情况下定义了 `CONFIG_DYNAMIC_DEBUG`)，否则它们会被编译。

### 函数接口

该 API 在以下内核代码中：

kernel/printk/printk.c  
include/linux/printk.h

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

**Original** Documentation/core-api/printk-formats.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

### 如何获得正确的 `printk` 格式占位符

作者 Randy Dunlap <[rdunlap@infradead.org](mailto:rdunlap@infradead.org)>

作者 Andrew Murray <[amurray@mpc-data.co.uk](mailto:amurray@mpc-data.co.uk)>

### 整数类型

若变量类型是 Type，则使用 `printk` 格式占位符。

char	%d 或 %x
unsigned char	%u 或 %x
short int	%d 或 %x
unsigned short int	%u 或 %x
int	%d 或 %x
unsigned int	%u 或 %x
long	%ld 或 %lx

unsigned long	%lu 或 %lx
long long	%lld 或 %llx
unsigned long long	%llu 或 %llx
size_t	%zu 或 %zx
ssize_t	%zd 或 %zx
s8	%d 或 %x
u8	%u 或 %x
s16	%d 或 %x
u16	%u 或 %x
s32	%d 或 %x
u32	%u 或 %x
s64	%lld 或 %llx
u64	%llu 或 %llx

如果 `<type>` 的大小依赖于配置选项 (例如 `sector_t`, `blkcnt_t`) 或其大小依赖于架构 (例如 `tcflag_t`), 则使用其可能的最大类型的格式占位符并显式强制转换为它。

例如

```
printf("test: sector number/total blocks: %llu/%llu\n",
       (unsigned long long)sector, (unsigned long long)blockcount);
```

提醒: `sizeof()` 返回类型为 `size_t`。

内核的 `printf` 不支持`%n`。显而易见, 浮点格式 (`%e`, `%f`, `%g`, `%a`) 也不被识别。使用任何不支持的占位符或长度限定符都会导致一个 `WARN` 并且终止 `vsnprintf()` 执行。

## 指针类型

一个原始指针值可以用`%p` 打印, 它将在打印前对地址进行哈希处理。内核也支持扩展占位符来打印不同类型的指针。

一些扩展占位符会打印给定地址上的数据, 而不是打印地址本身。在这种情况下, 以下错误消息可能会被打印出来, 而不是无法访问的消息:

```
(null)  data on plain NULL address
efault) data on invalid address
einval) invalid data on a valid address
```

## 普通指针

%p	abcdef12 or 00000000abcdef12
----	------------------------------

没有指定扩展名的指针（即没有修饰符的%p）被哈希（hash），以防止内核内存布局消息的泄露。这样还有一个额外的好处，就是提供一个唯一的标识符。在 64 位机器上，前 32 位被清零。当没有足够的熵进行散列处理时，内核将打印 (ptrval) 代替。

如果可能的话，使用专门的修饰符，如%pS 或%pB（如下所述），以避免打印一个必须事后解释的非哈希地址。如果不可能，而且打印地址的目的是为调试提供更多的消息，使用%p，并在调试过程中用 no\_hash\_pointers 参数启动内核，这将打印所有未修改的%p 地址。如果你真的想知道未修改的地址，请看下面的%px。

如果（也只有在）你将地址作为虚拟文件的内容打印出来，例如在 procfs 或 sysfs 中（使用 seq\_printf()，而不是 printk()）由用户空间进程读取，使用下面描述的%pK 修饰符，不要用%p 或%px。

## 错误指针

%pe	-ENOSPC
-----	---------

用于打印错误指针（即 IS\_ERR() 为真的指针）的符号错误名。不知道符号名的错误值会以十进制打印，而作为%pe 参数传递的非 ERR\_PTR 会被视为普通的%p。

## 符号/函数指针

%pS	versatile_init+0x0/0x110
%ps	versatile_init
%pSR	versatile_init+0x9/0x110 (with __builtin_extract_return_addr() translation)
%pB	prev_fn_of_versatile_init+0x88/0x88

S 和 s 占位符用于打印符号格式的指针。它们的结果是符号名称带有 (S) 或不带有 (s) 偏移量。如果禁用 KALLSYMS，则打印符号地址。

B 占位符的结果是带有偏移量的符号名，在打印堆栈回溯时应该使用。占位符将考虑编译器优化的影响，当使用尾部调用并使用 noreturn GCC 属性标记时，可能会发生这种优化。

如果指针在一个模块内，模块名称和可选的构建 ID 将被打印在符号名称之后，并在说明符的末尾添加一个额外的 b。

%pS	versatile_init+0x0/0x110 [module_name]
%pSb	versatile_init+0x0/0x110 [module_name ed5019fdf5e53be37cb1ba7899292d7e143b259e]
%pSRb	versatile_init+0x9/0x110 [module_name ed5019fdf5e53be37cb1ba7899292d7e143b259e] (with __builtin_extract_return_addr() translation)
%pBb	prev_fn_of_versatile_init+0x88/0x88 [module_name ed5019fdf5e53be37cb1ba7899292d7e143b259e]

## 来自 BPF / tracing 追踪的探查指针

```
%pks    kernel string
%pus    user string
```

k 和 u 指定符用于打印来自内核内存 (k) 或用户内存 (u) 的先前探测的内存。后面的 s 指定符的结果是打印一个字符串。对于直接在常规的 vsnprintf() 中使用时, (k) 和 (u) 注释被忽略, 但是, 当在 BPF 的 bpf\_trace\_printk() 之外使用时, 它会读取它所指向的内存, 不会出现错误。

## 内核指针

```
%pK    01234567 or 0123456789abcdef
```

用于打印应该对非特权用户隐藏的内核指针。%pK 的行为取决于 kptr\_restrict sysctl——详见 Documentation/admin-guide/sysctl/kernel.rst。

## 未经修改的地址

```
%px    01234567 or 0123456789abcdef
```

对于打印指针, 当你 真的想打印地址时。在用%px 打印指针之前, 请考虑你是否泄露了内核内存布局的敏感消息。%px 在功能上等同于%lx (或%lu)。%px 是首选, 因为它在 grep 查找时更唯一。如果将来我们需要修改内核处理打印指针的方式, 我们将能更好地找到调用点。

在使用%px 之前, 请考虑使用%p 并在调试过程中启用‘no\_hash\_pointer’‘内核参数是否足够 (参见上面的%p 描述)。%px 的一个有效场景可能是在 panic 发生之前立即打印消息, 这样无论如何都可以防止任何敏感消息被利用, 使用%px 就不需要用 no\_hash\_pointer 来重现 panic。

## 指针差异

```
%td    2560
%tx    a00
```

为了打印指针的差异, 使用 ptrdiff\_t 的%t 修饰符。

例如:

```
printk("test: difference between pointers: %td\n", ptr2 - ptr1);
```

## 结构体资源 (Resources)

```
%pr      [mem 0x60000000-0x6fffffff flags 0x2200] or
        [mem 0x0000000060000000-0x000000006fffffff flags 0x2200]
%pR     [mem 0x60000000-0x6fffffff pref] or
        [mem 0x0000000060000000-0x000000006fffffff pref]
```

用于打印结构体资源。R 和 r 占位符的结果是打印出的资源带有 (R) 或不带有 (r) 解码标志成员。  
通过引用传递。

## 物理地址类型 phys\_addr\_t

```
%pa[p] 0x01234567 or 0x0123456789abcdef
```

用于打印 phys\_addr\_t 类型 (以及它的衍生物, 如 resource\_size\_t), 该类型可以根据构建选项而变化, 无论 CPU 数据真实物理地址宽度如何。

通过引用传递。

## DMA 地址类型 dma\_addr\_t

```
%pad    0x01234567 or 0x0123456789abcdef
```

用于打印 dma\_addr\_t 类型, 该类型可以根据构建选项而变化, 而不考虑 CPU 数据路径的宽度。  
通过引用传递。

## 原始缓冲区为转义字符串

```
%*pE[achnops]
```

用于将原始缓冲区打印成转义字符串。对于以下缓冲区:

```
1b 62 20 5c 43 07 22 90 0d 5d
```

几个例子展示了如何进行转换 (不包括两端的引号)。:

```
%*pE          "\eb \C\aa"\220\r]"
%*pEhp       "\x1bb \C\x07"\x90\x0d]"
%*pEa        "\e\142\040\\103\aa\042\220\r\135"
```

转换规则是根据可选的标志组合来应用的 (详见:c:func:string\_escape\_mem 内核文档):

- a - ESCAPE\_ANY

- c - ESCAPE\_SPECIAL
- h - ESCAPE\_HEX
- n - ESCAPE\_NULL
- o - ESCAPE\_OCTAL
- p - ESCAPE\_NP
- s - ESCAPE\_SPACE

默认情况下，使用 ESCAPE\_ANY\_NP。

ESCAPE\_ANY\_NP 是许多情况下的明智选择，特别是对于打印 SSID。

如果字段宽度被省略，那么将只转义 1 个字节。

## 原始缓冲区为十六进制字符串

```
%*ph    00 01 02 ... 3f
%*phC   00:01:02: ... :3f
%*phD   00-01-02- ... -3f
%*phN   000102 ... 3f
```

对于打印小的缓冲区（最长 64 个字节），可以用一定的分隔符作为一个十六进制字符串。对于较大的缓冲区，可以考虑使用 `print_hex_dump()`。

## MAC/FDDI 地址

```
%pM    00:01:02:03:04:05
%pMR   05:04:03:02:01:00
%pMF   00-01-02-03-04-05
%pm    000102030405
%pmR   050403020100
```

用于打印以十六进制表示的 6 字节 MAC/FDDI 地址。`M` 和 `m` 占位符导致打印的地址有 (`M`) 或没有 (`m`) 字节分隔符。默认的字节分隔符是冒号 (`:`)。

对于 FDDI 地址，可以在 `M` 占位符之后使用 `F` 说明，以使用破折号 (—) 分隔符代替默认的分隔符。

对于蓝牙地址，`R` 占位符应使用在 `M` 占位符之后，以使用反转的字节顺序，适合于以小尾端顺序的蓝牙地址的肉眼可见的解析。

通过引用传递。

### IPv4 地址

```
%pI4    1.2.3.4  
%pi4   001.002.003.004  
%p[Ii]4[hndl]
```

用于打印 IPv4 点分隔的十进制地址。I4 和 i4 占位符的结果是打印的地址有 (i4) 或没有 (I4) 前导零。

附加的 h、n、b 和 l 占位符分别用于指定主机、网络、大尾端或小尾端地址。如果没有提供占位符，则使用默认的网络/大尾端顺序。

通过引用传递。

### IPv6 地址

```
%pI6    0001:0002:0003:0004:0005:0006:0007:0008  
%pi6   00010002000300040005000600070008  
%pI6c  1:2:3:4:5:6:7:8
```

用于打印 IPv6 网络顺序的 16 位十六进制地址。I6 和 i6 占位符的结果是打印的地址有 (I6) 或没有 (i6) 分号。始终使用前导零。

额外的 c 占位符可与 I 占位符一起使用，以打印压缩的 IPv6 地址，如 <https://tools.ietf.org/html/rfc5952> 所述

通过引用传递。

### IPv4/IPv6 地址 (generic, with port, flowinfo, scope)

```
%pIS    1.2.3.4          or 0001:0002:0003:0004:0005:0006:0007:0008  
%piS   001.002.003.004 or 00010002000300040005000600070008  
%pISc  1.2.3.4          or 1:2:3:4:5:6:7:8  
%pISpc 1.2.3.4:12345   or [1:2:3:4:5:6:7:8]:12345  
%p[Ii]S[pfschnbl]
```

用于打印一个 IP 地址，不需要区分它的类型是 AF\_INET 还是 AF\_INET6。一个指向有效结构体 sockaddr 的指针，通过 IS 或 ISc 指定，可以传递给这个格式占位符。

附加的 p、f 和 s 占位符用于指定 port(IPv4, IPv6)、flowinfo (IPv6) 和 scope(IPv6)。port 有一个 : 前缀，flowinfo 是 / 和范围是 %，每个后面都跟着实际的值。

对于 IPv6 地址，如果指定了额外的指定符 c，则使用 <https://tools.ietf.org/html/rfc5952> 描述的压缩 IPv6 地址。如 <https://tools.ietf.org/html/draft-ietf-6man-text-addr-representation-07> 所建议的，IPv6 地址由' [ ' ] ' 包围，以防止出现额外的占位符 p，f 或 s。

对于 IPv4 地址，也可以使用额外的 h，n，b 和 l 说明符，但对于 IPv6 地址则忽略。

通过引用传递。

更多例子:

%pISfc	1.2.3.4	or [1:2:3:4:5:6:7:8]/123456789
%pISSc	1.2.3.4	or [1:2:3:4:5:6:7:8]%1234567890
%pISpfc	1.2.3.4:12345	or [1:2:3:4:5:6:7:8]:12345/123456789

## UUID/GUID 地址

%pUb	00010203-0405-0607-0809-0a0b0c0d0e0f
%pUB	00010203-0405-0607-0809-0A0B0C0D0E0F
%pUL	03020100-0504-0706-0809-0a0b0c0e0e0f
%pUL	03020100-0504-0706-0809-0A0B0C0E0E0F

用于打印 16 字节的 UUID/GUIDs 地址。附加的 l , L , b 和 B 占位符用于指定小写 (l) 或大写 (L) 十六进制表示法中的小尾端顺序，以及小写 (b) 或大写 (B) 十六进制表示法中的大尾端顺序。

如果没有使用额外的占位符，则将打印带有小写十六进制表示法的默认大端顺序。

通过引用传递。

## 目录项 (dentry) 的名称

%pd{,2,3,4}
%pD{,2,3,4}

用于打印 dentry 名称；如果我们用 d\_move() 和它比较，名称可能是新旧混合的，但不会 oops。%pd dentry 比较安全，其相当于我们以前用的 %s dentry->d\_name.name，%pd<n> 打印 n 最后的组件。%pD 对结构文件做同样的事情。

通过引用传递。

## 块设备 (block\_device) 名称

%pg	sda, sda1 or loop0p1
-----	----------------------

用于打印 block\_device 指针的名称。

### **va\_format** 结构体

```
%pV
```

用于打印结构体 va\_format。这些结构包含一个格式字符串和 va\_list 如下

```
struct va_format {
    const char *fmt;
    va_list *va;
};
```

实现“递归 vsnprintf”。

如果没有一些机制来验证格式字符串和 va\_list 参数的正确性，请不要使用这个功能。

通过引用传递。

### 设备树节点

```
%p0F[fnpPcCF]
```

用于打印设备树节点结构。默认行为相当于%pOFF。

- f - 设备节点全称
- n - 设备节点名
- p - 设备节点句柄
- P - 设备节点路径规范 (名称 +@ 单位)
- F - 设备节点标志
- c - 主要兼容字符串
- C - 全兼容字符串

当使用多个参数时，分隔符是’ :’。

例如

%p0F /foo/bar@0	- Node full name
%p0Ff /foo/bar@0	- Same as above
%p0Ffp /foo/bar@0:10	- Node full name + phandle
%p0FfcF /foo/bar@0:foo,device:--P-	- Node full name + major compatible string + node flags D - dynamic d - detached P - Populated B - Populated bus

通过引用传递。

## Fwnode handles

```
%pfw[fP]
```

用于打印 fwnode\_handles 的消息。默认情况下是打印完整的节点名称，包括路径。这些修饰符在功能上等同于上面的%pOF。

- f - 节点的全名，包括路径。
- P - 节点名称，包括地址（如果有的话）。

例如 (ACPI)

%pfwf \_SB.PCI0.CI02.port@1.endpoint@0	- Full node name
%pfp endpoint@0	- Node name

例如 (OF)

%pfwf /ocp@68000000/i2c@48072000/camera@10/port/endpoint	- Full name
%pfp endpoint	- Node name

## 时间和日期

%pt[RT]	YYYY-mm-ddTHH:MM:SS
%pt[RT]s	YYYY-mm-dd HH:MM:SS
%pt[RT]d	YYYY-mm-dd
%pt[RT]t	HH:MM:SS
%pt[RT][dt][r][s]	

用于打印日期和时间：

R struct rtc_time structure
T time64_t type

以我们（人类）可读的格式。

默认情况下，年将以 1900 为单位递增，月将以 1 为单位递增。使用%pt[RT]r (raw) 来抑制这种行为。

%pt[RT]s (空格) 将覆盖 ISO 8601 的分隔符，在日期和时间之间使用' ' (空格) 而不是' T' (大写 T)。当日期或时间被省略时，它不会有任何影响。

通过引用传递。

## clk 结构体

```
%pC    pLL1
%pCn   pLL1
```

用于打印 clk 结构。%pC 和%pCn 打印时钟的名称（通用时钟框架）或唯一的 32 位 ID（传统时钟框架）。通过引用传递。

## 位图及其衍生物，如 cpumask 和 nodemask

```
%*pb    0779
%*pbl   0,3-6,8-10
```

对于打印位图(bitmap) 及其派生的 cpumask 和 nodemask, %\*pb 输出以字段宽度为位数的位图, %\*pbl 输出以字段宽度为位数的范围列表。

字段宽度用值传递, 位图用引用传递。可以使用辅助宏 cpumask\_pr\_args() 和 nodemask\_pr\_args() 来方便打印 cpumask 和 nodemask。

## 标志位字段，如页标志、gfp\_flags

```
%pGp   referenced|uptodate|lru|active|private|node=0|zone=2|lastcpupid=0xffffffff
%pGg   GFP_USER|GFP_DMA32|GFP_NOWARN
%pGv   read|exec|mayread|maywrite|mayexec|denywrite
```

将 flags 位字段打印为构造值的符号常量集合。标志的类型由第三个字符给出。目前支持的是 [p]age flags, [v]ma\_flags(都期望 unsigned long \*) 和 [g]fp\_flags(期望 gfp\_t \*)。标志名称和打印顺序取决于特定的类型。

注意, 这种格式不应该直接用于跟踪点的:c:func:TP\_printk() 部分。相反, 应使用 <trace/events/mmflags.h> 中的 show\_\*\_flags() 函数。

通过引用传递。

## 网络设备特性

```
%pNF    0x000000000000c000
```

用于打印 netdev\_features\_t。

通过引用传递。

## V4L2 和 DRM FourCC 代码 (像素格式)

```
%p4cc
```

打印 V4L2 或 DRM 使用的 FourCC 代码，包括格式端序及其十六进制的数值。

通过引用传递。

例如：

```
%p4cc BG12 little-endian (0x32314742)
%p4cc Y10 little-endian (0x20303159)
%p4cc NV12 big-endian (0xb231564e)
```

## 谢谢

如果您添加了其他%p 扩展，请在可行的情况下，用一个或多个测试用例扩展 <lib/test\_printf.c>。

谢谢你的合作和关注。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/core-api/workqueue.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## 并发管理的工作队列 (cmwq)

日期 September, 2010

作者 Tejun Heo <[tj@kernel.org](mailto:tj@kernel.org)>

作者 Florian Mickler <[florian@mickler.org](mailto:florian@mickler.org)>

### 简介

在很多情况下，需要一个异步进程的执行环境，工作队列（wq）API 是这种情况下最常用的机制。

当需要这样一个异步执行上下文时，一个描述将要执行的函数的工作项（work，即一个待执行的任务）被放在队列中。一个独立的线程作为异步执行环境。该队列被称为 workqueue，线程被称为工作者（worker，即执行这一队列的线程）。

当工作队列上有工作项时，工作者会一个接一个地执行与工作项相关的函数。当工作队列中没有任何工作项时，工作者就会变得空闲。当一个新的工作项被排入队列时，工作者又开始执行。

### 为什么要 cmwq？

在最初的 wq 实现中，多线程（MT）wq 在每个 CPU 上有一个工作者线程，而单线程（ST）wq 在全系统有一个工作者线程。一个 MT wq 需要保持与 CPU 数量相同的工作者数量。这些年来，内核增加了很多 MT wq 的用户，随着 CPU 核心数量的不断增加，一些系统刚启动就达到了默认的 32k PID 的饱和空间。

尽管 MT wq 浪费了大量的资源，但所提供的并发性水平却不能令人满意。这个限制在 ST 和 MT wq 中都有，只是在 MT 中没有那么严重。每个 wq 都保持着自己独立的工作者池。一个 MT wq 只能为每个 CPU 提供一个执行环境，而一个 ST wq 则为整个系统提供一个。工作项必须竞争这些非常有限的执行上下文，从而导致各种问题，包括在单一执行上下文周围容易发生死锁。

(MT wq) 所提供的并发性水平和资源使用之间的矛盾也迫使其用户做出不必要的权衡，比如 libata 选择使用 ST wq 来轮询 PIO，并接受一个不必要的限制，即没有两个轮询 PIO 可以同时进行。由于 MT wq 并没有提供更好的并发性，需要更高层次的并发性的用户，如 async 或 fscache，不得不实现他们自己的线程池。

并发管理工作队列（cmwq）是对 wq 的重新实现，重点是以下目标。

- 保持与原始工作队列 API 的兼容性。
- 使用由所有 wq 共享的每 CPU 统一的工作者池，在不浪费大量资源的情况下按需提供灵活的并发水平。
- 自动调节工作者池和并发水平，使 API 用户不需要担心这些细节。

### 设计

为了简化函数的异步执行，引入了一个新的抽象概念，即工作项。

一个工作项是一个简单的结构，它持有一个指向将被异步执行的函数的指针。每当一个驱动程序或子系统希望一个函数被异步执行时，它必须建立一个指向该函数的工作项，并在工作队列中排队等待该工作项。（就是挂到 workqueue 队列里面去）

特定目的线程，称为工作线程（工作者），一个接一个地执行队列中的功能。如果没有工作项排队，工作者线程就会闲置。这些工作者线程被管理在所谓的工作者池中。

cmwq 设计区分了面向用户的工作队列，子系统和驱动程序在上面排队工作，以及管理工作者池和处理排队工作项的后端机制。

每个可能的 CPU 都有两个工作者池，一个用于正常的工作项，另一个用于高优先级的工作项，还有一些额外的工作者池，用于服务未绑定工作队列的工作项目——这些后备池的数量是动态的。

当他们认为合适的时候，子系统和驱动程序可以通过特殊的 `workqueue API` 函数创建和排队工作项。他们可以通过在工作队列上设置标志来影响工作项执行方式的某些方面，他们把工作项放在那里。这些标志包括诸如 CPU 定位、并发限制、优先级等等。要获得详细的概述，请参考下面的 `alloc_workqueue()` 的 API 描述。

当一个工作项被排入一个工作队列时，目标工作池将根据队列参数和工作队列属性确定，并被附加到工作池的共享工作列表上。例如，除非特别重写，否则一个绑定的工作队列的工作项将被排在与发起线程运行的 CPU 相关的普通或高级工作工作者池的工作项列表中。

对于任何工作者池的实施，管理并发水平（有多少执行上下文处于活动状态）是一个重要问题。最低水平是为了节省资源，而饱和水平是指系统被充分使用。

每个与实际 CPU 绑定的 worker-pool 通过钩住调度器来实现并发管理。每当一个活动的工作者被唤醒或睡眠时，工作者池就会得到通知，并跟踪当前可运行的工作者的数量。一般来说，工作项不会占用 CPU 并消耗很多周期。这意味着保持足够的并发性以防止工作处理停滞应该是最优的。只要 CPU 上有一个或多个可运行的工作者，工作者池就不会开始执行新的工作，但是，当最后一个运行的工作者进入睡眠状态时，它会立即安排一个新的工作者，这样 CPU 就不会在有待处理的工作项目时闲置。这允许在不损失执行带宽的情况下使用最少的工作者。

除了 `kthreads` 的内存空间外，保留空闲的工作者并没有其他成本，所以 `cmwq` 在杀死它们之前会保留一段时间的空闲。

对于非绑定的工作队列，后备池的数量是动态的。可以使用 `apply_workqueue_attrs()` 为非绑定工作队列分配自定义属性，`workqueue` 将自动创建与属性相匹配的后备工作者池。调节并发水平的责任在用户身上。也有一个标志可以将绑定的 `wq` 标记为忽略并发管理。详情请参考 API 部分。

前进进度的保证依赖于当需要更多的执行上下文时可以创建工作，这也是通过使用救援工作者来保证的。所有可能在处理内存回收的代码路径上使用的工作项都需要在 `wq` 上排队，`wq` 上保留了一个救援工作者，以便在内存有压力的情况下执行。否则，工作者池就有可能出现死锁，等待执行上下文释放出来。

## 应用程序编程接口 (API)

`alloc_workqueue()` 分配了一个 `wq`。原来的 `create_*workqueue()` 函数已被废弃，并计划删除。`alloc_workqueue()` 需要三个参数 - `@name`，`@flags` 和 `@max_active`。`@name` 是 `wq` 的名称，如果有的话，也用作救援线程的名称。

一个 `wq` 不再管理执行资源，而是作为前进进度保证、刷新 (`flush`) 和工作项属性的域。`@flags` 和 `@max_active` 控制着工作项如何被分配执行资源、安排和执行。

### flags

**WQ\_UNBOUND** 排队到非绑定 wq 的工作项由特殊的工作者池提供服务，这些工作者不绑定在任何特定的 CPU 上。这使得 wq 表现得像一个简单的执行环境提供者，没有并发管理。非绑定工作者池试图尽快开始执行工作项。非绑定的 wq 牺牲了局部性，但在以下情况下是有用的。

- 预计并发水平要求会有很大的波动，使用绑定的 wq 最终可能会在不同的 CPU 上产生大量大部分未使用的工作者，因为发起线程在不同的 CPU 上跳转。
- 长期运行的 CPU 密集型工作负载，可以由系统调度器更好地管理。

**WQ\_FREEZABLE** 一个可冻结的 wq 参与了系统暂停操作的冻结阶段。wq 上的工作项被排空，在解冻之前没有新的工作项开始执行。

**WQ\_MEM\_RECLAIM** 所有可能在内存回收路径中使用的 wq 都必须设置这个标志。无论内存压力如何，wq 都能保证至少有一个执行上下文。

**WQ\_HIGHPRI** 高优先级 wq 的工作项目被排到目标 cpu 的高优先级工作者池中。高优先级的工作者池由具有较高级别的工作者线程提供服务。

请注意，普通工作者池和高优先级工作者池之间并不相互影响。他们各自维护其独立的工作者池，并在其工作者之间实现并发管理。

**WQ\_CPU\_INTENSIVE** CPU 密集型 wq 的工作项对并发水平没有贡献。换句话说，可运行的 CPU 密集型工作项不会阻止同一工作者池中的其他工作项开始执行。这对于那些预计会占用 CPU 周期的绑定工作项很有用，这样它们的执行就会受到系统调度器的监管。

尽管 CPU 密集型工作项不会对并发水平做出贡献，但它们的执行开始仍然受到并发管理的管制，可运行的非 CPU 密集型工作项会延迟 CPU 密集型工作项的执行。

这个标志对于未绑定的 wq 来说是没有意义的。

请注意，标志 **WQ\_NON\_REENTRANT** 不再存在，因为现在所有的工作队列都是不可逆的——任何工作项都保证在任何时间内最多被整个系统的一个工作者执行。

### max\_active

@max\_active 决定了每个 CPU 可以分配给 wq 的工作项的最大执行上下文数量。例如，如果 @max\_active 为 16，每个 CPU 最多可以同时执行 16 个 wq 的工作项。

目前，对于一个绑定的 wq，@max\_active 的最大限制是 512，当指定为 0 时使用的默认值是 256。对于非绑定的 wq，其限制是 512 和 4 \* num\_possible\_cpus() 中的较高值。这些值被选得足够高，所以它们不是限制性因素，同时会在失控情况下提供保护。

一个 wq 的活动工作项的数量通常由 wq 的用户来调节，更具体地说，是由用户在同一时间可以排列多少个工作项来调节。除非有特定的需求来控制活动工作项的数量，否则建议指定为“0”。

一些用户依赖于 ST wq 的严格执行顺序。@max\_active 为 1 和 WQ\_UNBOUND 的组合用来实现这种行为。这种 wq 上的工作项目总是被排到未绑定的工作池中，并且在任何时候都只有一个工作项目处于活动状态，从而实

现与 ST wq 相同的排序属性。

在目前的实现中，上述配置只保证了特定 NUMA 节点内的 ST 行为。相反，`alloc_ordered_queue()` 应该被用来实现全系统的 ST 行为。

## 执行场景示例

下面的示例执行场景试图说明 cmwq 在不同配置下的行为。

工作项 w0、w1、w2 被排到同一个 CPU 上的一个绑定的 wq q0 上。w0 消耗 CPU 5ms，然后睡眠 10ms，然后在完成之前再次消耗 CPU 5ms。

忽略所有其他的任务、工作和处理开销，并假设简单的 FIFO 调度，下面是一个高度简化的原始 wq 的可能事件序列的版本。：

TIME IN MSECS	EVENT
0	w0 starts and burns CPU
5	w0 sleeps
15	w0 wakes up and burns CPU
20	w0 finishes
20	w1 starts and burns CPU
25	w1 sleeps
35	w1 wakes up and finishes
35	w2 starts and burns CPU
40	w2 sleeps
50	w2 wakes up and finishes

And with cmwq with `@max_active >= 3`,

TIME IN MSECS	EVENT
0	w0 starts and burns CPU
5	w0 sleeps
5	w1 starts and burns CPU
10	w1 sleeps
10	w2 starts and burns CPU
15	w2 sleeps
15	w0 wakes up and burns CPU
20	w0 finishes
20	w1 wakes up and finishes
25	w2 wakes up and finishes

如果 `@max_active == 2`,

TIME IN MSECS	EVENT
0	w0 starts and burns CPU
5	w0 sleeps
5	w1 starts and burns CPU
10	w1 sleeps
15	w0 wakes up and burns CPU
20	w0 finishes

20	w1 wakes up and finishes
20	w2 starts and burns CPU
25	w2 sleeps
35	w2 wakes up and finishes

现在，我们假设 w1 和 w2 被排到了不同的 wq q1 上，这个 wq q1 有 WQ\_CPU\_INTENSIVE 设置：

TIME IN MSECS	EVENT
0	w0 starts and burns CPU
5	w0 sleeps
5	w1 and w2 start and burn CPU
10	w1 sleeps
15	w2 sleeps
15	w0 wakes up and burns CPU
20	w0 finishes
20	w1 wakes up and finishes
25	w2 wakes up and finishes

## 指南

- 如果一个 wq 可能处理在内存回收期间使用的工作项目，请不要忘记使用 WQ\_MEM\_RECLAIM。每个设置了 WQ\_MEM\_RECLAIM 的 wq 都有一个为其保留的执行环境。如果在内存回收过程中使用的多个工作项之间存在依赖关系，它们应该被排在不同的 wq 中，每个 wq 都有 WQ\_MEM\_RECLAIM。
- 除非需要严格排序，否则没有必要使用 ST wq。
- 除非有特殊需要，建议使用 0 作为 @max\_active。在大多数使用情况下，并发水平通常保持在默认限制之下。
- 一个 wq 作为前进进度保证 (WQ\_MEM\_RECLAIM, 冲洗 (flush) 和工作项属性的域。不涉及内存回收的工作项，不需要作为工作项组的一部分被刷新，也不需要任何特殊属性，可以使用系统中的一个 wq。使用专用 wq 和系统 wq 在执行特性上没有区别。
- 除非工作项预计会消耗大量的 CPU 周期，否则使用绑定的 wq 通常是有益的，因为 wq 操作和工作项执行中的定位水平提高了。

## 调试

因为工作函数是由通用的工作者线程执行的，所以需要一些手段来揭示一些行为不端的工作队列用户。

工作者线程在进程列表中显示为：

root	5671	0.0	0.0	0	0 ?	S	12:07	0:00	[kworker/0:1]
root	5672	0.0	0.0	0	0 ?	S	12:07	0:00	[kworker/1:2]
root	5673	0.0	0.0	0	0 ?	S	12:12	0:00	[kworker/0:0]
root	5674	0.0	0.0	0	0 ?	S	12:13	0:00	[kworker/1:0]

如果 kworkers 失控了（使用了太多的 cpu），有两类可能的问题：

1. 正在迅速调度的事情
2. 一个消耗大量 cpu 周期的工作项。

第一个可以用追踪的方式进行跟踪：

```
$ echo workqueue:workqueue_queue_work > /sys/kernel/debug/tracing/set_event
$ cat /sys/kernel/debug/tracing/trace_pipe > out.txt
(wait a few secs)
```

如果有什么东西在工作队列上忙着做循环，它就会主导输出，可以用工作项函数确定违规者。

对于第二类问题，应该可以只检查违规工作者线程的堆栈跟踪。

```
$ cat /proc/THE_OFFENDING_KWORKER/stack
```

工作项函数在堆栈追踪中应该是微不足道的。

## 内核内联文档参考

该 API 在以下内核代码中：

include/linux/workqueue.h  
kernel/workqueue.c

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/core-api/symbol-namespaces.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## 符号命名空间 (Symbol Namespaces)

本文档描述了如何使用符号命名空间来构造通过 EXPORT\_SYMBOL() 系列宏导出的内核内符号的导出面。

### 1. 简介

符号命名空间已经被引入，作为构造内核内 API 的导出面的一种手段。它允许子系统维护者将他们导出的符号划分进独立的命名空间。这对于文档的编写非常有用（想想 SUBSYSTEM\_DEBUG 命名空间），也可以限制一组符号在内核其他部分的使用。今后，使用导出到命名空间的符号的模块必须导入命名空间。否则，内核将根据其配置，拒绝加载该模块或警告说缺少导入。

### 2. 如何定义符号命名空间

符号可以用不同的方法导出到命名空间。所有这些都在改变 EXPORT\_SYMBOL 和与之类似的一些宏被检测到的方式，以创建 ksymtab 条目。

#### 2.1 使用 EXPORT\_SYMBOL 宏

除了允许将内核符号导出到内核符号表的宏 EXPORT\_SYMBOL() 和 EXPORT\_SYMBOL\_GPL() 之外，这些宏的变体还可以将符号导出到某个命名空间：EXPORT\_SYMBOL\_NS() 和 EXPORT\_SYMBOL\_NS\_GPL()。它们需要一个额外的参数：命名空间（the namespace）。请注意，由于宏扩展，该参数需要是一个预处理器符号。例如，要把符号 `usb_stor_suspend` 导出到命名空间 `USB_STORAGE`，请使用：

```
EXPORT_SYMBOL_NS(usb_stor_suspend, USB_STORAGE);
```

相应的 ksymtab 条目结构体 `kernel_symbol` 将有相应的成员 命名空间集。导出时未指明命名空间的符号将指向 `NULL`。如果没有定义命名空间，则默认没有。`modpost` 和 `kernel/module/main.c` 分别在构建时或模块加载时使用名称空间。

#### 2.2 使用 DEFAULT\_SYMBOL\_NAMESPACE 定义

为一个子系统的所有符号定义命名空间可能会非常冗长，并可能变得难以维护。因此，我们提供了一个默认定义 (`DEFAULT_SYMBOL_NAMESPACE`)，如果设置了这个定义，它将成为所有没有指定命名空间的 EXPORT\_SYMBOL() 和 EXPORT\_SYMBOL\_GPL() 宏扩展的默认定义。

有多种方法来指定这个定义，使用哪种方法取决于子系统和维护者的喜好。第一种方法是在子系统的 `Makefile` 中定义默认命名空间。例如，如果要将 `usb-common` 中定义的所有符号导出到 `USB_COMMON` 命名空间，可以在 `drivers/usb/common/Makefile` 中添加这样一行：

```
ccflags-y += -DDEFAULT_SYMBOL_NAMESPACE=USB_COMMON
```

这将影响所有 EXPORT\_SYMBOL() 和 EXPORT\_SYMBOL\_GPL() 语句。当这个定义存在时，用 EXPORT\_SYMBOL\_NS() 导出的符号仍然会被导出到作为命名空间参数传递的命名空间中，因为这个参数优先于默认的符号命名空间。

定义默认命名空间的第二个选项是直接在编译单元中作为预处理声明。上面的例子就会变成：

```
#undef DEFAULT_SYMBOL_NAMESPACE
#define DEFAULT_SYMBOL_NAMESPACE USB_COMMON
```

应置于相关编译单元中任何 EXPORT\_SYMBOL 宏之前

### 3. 如何使用命名空间中导出的符号

为了使用被导出到命名空间的符号，内核模块需要明确地导入这些命名空间。否则内核可能会拒绝加载该模块。模块代码需要使用宏 MODULE\_IMPORT\_NS 来表示它所使用的命名空间的符号。例如，一个使用 usb\_stor\_suspend 符号的模块，需要使用如下语句导入命名空间 USB\_STORAGE：

```
MODULE_IMPORT_NS(USB_STORAGE);
```

这将在模块中为每个导入的命名空间创建一个 modinfo 标签。这也顺带使得可以用 modinfo 检查模块已导入的命名空间：

```
$ modinfo drivers/usb/storage/ums-karma.ko
[...]
import_ns:      USB_STORAGE
[...]
```

建议将 MODULE\_IMPORT\_NS() 语句添加到靠近其他模块元数据定义的地方，如 MODULE\_AUTHOR() 或 MODULE\_LICENSE()。关于自动创建缺失的导入语句的方法，请参考第 5 节。

### 4. 加载使用命名空间符号的模块

在模块加载时（比如 insmod），内核将检查每个从模块中引用的符号是否可用，以及它可能被导出到的名字空间是否被模块导入。内核的默认行为是拒绝加载那些没有指明足以导入的模块。此错误会被记录下来，并且加载将以 EINVAL 方式失败。要允许加载不满足这个前提条件的模块，可以使用此配置选项：设置 MODULE\_ALLOW\_MISSING\_NAMESPACE\_IMPORTS=y 将使加载不受影响，但会发出警告。

## 5. 自动创建 MODULE\_IMPORT\_NS 声明

缺少命名空间的导入可以在构建时很容易被检测到。事实上，如果一个模块使用了一个命名空间的符号而没有导入它，modpost 会发出警告。MODULE\_IMPORT\_NS() 语句通常会被添加到一个明确的位置（和其他模块元数据一起）。为了使模块作者（和子系统维护者）的生活更加轻松，我们提供了一个脚本和 make 目标来修复丢失的导入。修复丢失的导入可以用：

```
$ make nsdeps
```

对模块作者来说，以下情况可能很典型：

- 编写依赖未导入命名空间的符号的代码
- ``make``
- 注意 ``modpost`` 的警告，提醒你有一个丢失的导入。
- 运行 ``make nsdeps`` 将导入添加到正确的代码位置。

对于引入命名空间的子系统维护者来说，其步骤非常相似。同样，make nsdeps 最终将为树内模块添加缺失的命名空间导入：

- 向命名空间转移或添加符号（例如，使用 EXPORT\_SYMBOL\_NS()）。
- `make e`（最好是用 allmodconfig 来覆盖所有的内核模块）。
- 注意 ``modpost`` 的警告，提醒你有一个丢失的导入。
- 运行 ``maknsdeps`` 将导入添加到正确的代码位置。

你也可以为外部模块的构建运行 nsdeps。典型的用法是：

```
$ make -C <path_to_kernel_src> M=$PWD nsdeps
```

## 数据结构和低级实用程序

在整个内核中使用的函数库。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/core-api/kobject.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## 关于 kobjects、ksets 和 ktypes 的一切你没想过需要了解的东西

作者 Greg Kroah-Hartman <[gregkh@linuxfoundation.org](mailto:gregkh@linuxfoundation.org)>

最后一次更新 December 19, 2007

根据 Jon Corbet 于 2003 年 10 月 1 日为 lwn.net 撰写的原创文章改编，网址是：<https://lwn.net/Articles/51437/>

理解驱动模型和建立在其上的 kobject 抽象的部分的困难在于，没有明显的切入点。处理 kobjects 需要理解一些不同的类型，所有这些类型都会相互引用。为了使事情变得更简单，我们将多路并进，从模糊的术语开始，并逐渐增加细节。那么，先来了解一些我们将要使用的术语的简明定义吧。

- 一个 kobject 是一个 kobject 结构体类型的对象。Kobjects 有一个名字和一个引用计数。一个 kobject 也有一个父指针（允许对象被排列成层次结构），一个特定的类型，并且，通常在 sysfs 虚拟文件系统中表示。

Kobjects 本身通常并不引人关注；相反它们常常被嵌入到其他包含真正引人注目的代码的结构体中。

任何结构体都 不应该 有一个以上的 kobject 嵌入其中。如果有的话，对象的引用计数肯定会被打乱，而且不正确，你的代码就会出现错误。所以不要这样做。

- ktype 是嵌入一个 kobject 的对象的类型。每个嵌入 kobject 的结构体都需要一个相应的 ktype。ktype 控制着 kobject 在被创建和销毁时的行为。
- 一个 kset 是一组 kobjects。这些 kobjects 可以是相同的 ktype 或者属于不同的 ktype。kset 是 kobjects 集合的基本容器类型。Ksets 包含它们自己的 kobjects，但你可以安全地忽略这个实现细节，因为 kset 的核心代码会自动处理这个 kobject。

当你看到一个下面全是其他目录的 sysfs 目录时，通常这些目录中的每一个都对应于同一个 kset 中的一个 kobject。

我们将研究如何创建和操作所有这些类型。将采取一种自下而上的方法，所以我们将回到 kobjects。

### 嵌入 kobjects

内核代码很少创建孤立的 kobject，只有一个主要的例外，下面会解释。相反，kobjects 被用来控制对一个更大的、特定领域的对象的访问。为此，kobjects 会被嵌入到其他结构中。如果你习惯于用面向对象的术语来思考问题，那么 kobjects 可以被看作是一个顶级的抽象类，其他的类都是从它派生出来的。一个 kobject 实现了一系列的功能，这些功能本身并不特别有用，但在其他对象中却很好用。C 语言不允许直接表达继承，所以必须使用其他技术——比如结构体嵌入。

（对于那些熟悉内核链表实现的人来说，这类似于“list\_head”结构本身很少有用，但总是被嵌入到感兴趣的更大的对象中）。

例如，`drivers/uio/uio.c` 中的 IO 代码有一个结构体，定义了与 uio 设备相关的内存区域：

```
struct uio_map {
    struct kobject kobj;
    struct uio_mem *mem;
};
```

如果你有一个 `uio_map` 结构体，找到其嵌入的 `kobject` 只是一个使用 `kobj` 成员的问题。然而，与 `kobjects` 一起工作的代码往往遇到相反的问题：给定一个结构体 `kobject` 的指针，指向包含结构体的指针是什么？你必须避免使用一些技巧（比如假设 `kobject` 在结构的开头），相反，你得使用 `container_of()` 宏，其可以在 `<linux/kernel.h>` 中找到：

```
container_of(ptr, type, member)
```

其中：

- `ptr` 是一个指向嵌入 `kobject` 的指针，
- `type` 是包含结构体的类型，
- `member` 是指针所指向的结构体域的名称。

`container_of()` 的返回值是一个指向相应容器类型的指针。因此，例如，一个嵌入到 `uio_map` 结构中的 `kobject` 结构体的指针 `kp` 可以被转换为一个指向 包含 `uio_map` 结构体的指针，方法是：

```
struct uio_map *u_map = container_of(kp, struct uio_map, kobj);
```

为了方便起见，程序员经常定义一个简单的宏，用于将 `kobject` 指针 反推 到包含类型。在早期的 `drivers/uio/uio.c` 中正是如此，你可以在这里看到：

```
struct uio_map {
    struct kobject kobj;
    struct uio_mem *mem;
};

#define to_map(map) container_of(map, struct uio_map, kobj)
```

其中宏的参数 “`map`” 是一个指向有关的 `kobject` 结构体的指针。该宏随后被调用：

```
struct uio_map *map = to_map(kobj);
```

### kobjects 的初始化

当然，创建 `kobject` 的代码必须初始化该对象。一些内部字段是通过（强制）调用 `kobject_init()` 来设置的：

```
void kobject_init(struct kobject *kobj, struct kobj_type *ktype);
```

`ktype` 是正确创建 `kobject` 的必要条件，因为每个 `kobject` 都必须有一个相关的 `kobj_type`。在调用 `kobject_init()` 后，为了向 `sysfs` 注册 `kobject`，必须调用函数 `kobject_add()`：

```
int kobject_add(struct kobject *kobj, struct kobject *parent,
                const char *fmt, ...);
```

这将正确设置 kobject 的父级和 kobject 的名称。如果该 kobject 要与一个特定的 kset 相关联，在调用 kobject\_add() 之前必须分配 kobj->kset。如果 kset 与 kobject 相关联，则 kobject 的父级可以在调用 kobject\_add() 时被设置为 NULL，则 kobject 的父级将是 kset 本身。

由于 kobject 的名字是在它被添加到内核时设置的，所以 kobject 的名字不应该被直接操作。如果你必须改变 kobject 的名字，请调用 kobject\_rename():

```
int kobject_rename(struct kobject *kobj, const char *new_name);
```

kobject\_rename() 函数不会执行任何锁定操作，也不会对 name 进行可靠性检查，所以调用者自己检查和串行化操作是明智的选择

有一个叫 kobject\_set\_name() 的函数，但那是历史遗产，正在被删除。如果你的代码需要调用这个函数，那么它是不正确的，需要被修复。

要正确访问 kobject 的名称，请使用函数 kobject\_name():

```
const char *kobject_name(const struct kobject *kobj);
```

有一个辅助函数可以同时初始化和添加 kobject 到内核中，令人惊讶的是，该函数被称为 kobject\_init\_and\_add():

```
int kobject_init_and_add(struct kobject *kobj, struct kobj_type *ktype,
                        struct kobject *parent, const char *fmt, ...);
```

参数与上面描述的单个 kobject\_init() 和 kobject\_add() 函数相同。

## Uevents

当一个 kobject 被注册到 kobject 核心后，你需要向全世界宣布它已经被创建了。这可以通过调用 kobject\_uevent() 来实现：

```
int kobject_uevent(struct kobject *kobj, enum kobject_action action);
```

当 kobject 第一次被添加到内核时，使用 *KOBJ\_ADD* 动作。这应该在该 kobject 的任何属性或子对象被正确初始化后进行，因为当这个调用发生时，用户空间会立即开始寻找它们。

当 kobject 从内核中移除时（关于如何做的细节在下面），**KOBJ\_REMOVE** 的 uevent 将由 kobject 核心自动创建，所以调用者不必担心手动操作。

## 引用计数

kobject 的关键功能之一是作为它所嵌入的对象的一个引用计数器。只要对该对象的引用存在，该对象（以及支持它的代码）就必须继续存在。用于操作 kobject 的引用计数的低级函数是：

```
struct kobject *kobject_get(struct kobject *kobj);
void kobject_put(struct kobject *kobj);
```

对 kobject\_get() 的成功调用将增加 kobject 的引用计数器值并返回 kobject 的指针。

当引用被释放时，对 kobject\_put() 的调用将递减引用计数值，并可能释放该对象。请注意，kobject\_init() 将引用计数设置为 1，所以设置 kobject 的代码最终需要 kobject\_put() 来释放该引用。

因为 kobjects 是动态的，所以它们不能以静态方式或在堆栈中声明，而总是以动态方式分配。未来版本的内核将包含对静态创建的 kobjects 的运行时检查，并将警告开发者这种不当的使用。

如果你使用 struct kobject 只是为了给你的结构体提供一个引用计数器，请使用 struct kref 来代替；kobject 是多余的。关于如何使用 kref 结构体的更多信息，请参见 Linux 内核源代码树中的文件 Documentation/core-api/kref.rst

## 创建“简单的”kobjects

有时，开发者想要的只是在 sysfs 层次结构中创建一个简单的目录，而不必去搞那些复杂的 ksets、显示和存储函数，以及其他细节。这是一个应该创建单个 kobject 的例外。要创建这样一个条目（即简单的目录），请使用函数：

```
struct kobject *kobject_create_and_add(const char *name, struct kobject *parent);
```

这个函数将创建一个 kobject，并将其放在 sysfs 中指定的父 kobject 下面的位置。要创建与此 kobject 相关的简单属性，请使用：

```
int sysfs_create_file(struct kobject *kobj, const struct attribute *attr);
```

或者：

```
int sysfs_create_group(struct kobject *kobj, const struct attribute_group *grp);
```

这里使用的两种类型的属性，与已经用 kobject\_create\_and\_add() 创建的 kobject，都可以是 kobj\_attribute 类型，所以不需要创建特殊的自定义属性。

参见示例模块，samples/kobject/kobject-example.c，以了解一个简单的 kobject 和属性的实现。

## ktypes 和释放方法

以上讨论中还缺少一件重要的事情，那就是当一个 kobject 的引用次数达到零的时候会发生什么。创建 kobject 的代码通常不知道何时会发生这种情况；首先，如果它知道，那么使用 kobject 就没有什么意义。当 sysfs 被引入时，即使是可预测的对象生命周期也会变得更加复杂，因为内核的其他部分可以获得在系统中注册的任何 kobject 的引用。

最终的结果是，一个由 kobject 保护的结构体在其引用计数归零之前不能被释放。引用计数不受创建 kobject 的代码的直接控制。因此，每当它的一个 kobjects 的最后一个引用消失时，必须异步通知该代码。

一旦你通过 kobject\_add() 注册了你的 kobject，你绝对不能使用 kfree() 来直接释放它。唯一安全的方法是使用 kobject\_put()。在 kobject\_init() 之后总是使用 kobject\_put() 以避免错误的发生是一个很好的做法。这个通知是通过 kobject 的 release() 方法完成的。通常这样的方法有如下形式：

```
void my_object_release(struct kobject *kobj)
{
    struct my_object *mine = container_of(kobj, struct my_object, kobj);

    /* Perform any additional cleanup on this object, then... */
    kfree(mine);
}
```

有一点很重要：每个 kobject 都必须有一个 release() 方法，而且这个 kobject 必须持续存在（处于一致的状态），直到这个方法被调用。如果这些约束条件没有得到满足，那么代码就是有缺陷的。注意，如果你忘记提供 release() 方法，内核会警告你。不要试图通过提供一个“空”的释放函数来摆脱这个警告。

如果你的清理函数只需要调用 kfree()，那么你必须创建一个包装函数，该函数使用 container\_of() 来向上造型到正确的类型（如上面的例子所示），然后在整个结构体上调用 kfree()。

注意，kobject 的名字在 release 函数中是可用的，但它不能在这个回调中被改变。否则，在 kobject 核心中会出现内存泄漏，这让人很不爽。

有趣的是，release() 方法并不存储在 kobject 本身；相反，它与 ktype 相关。因此，让我们引入结构体 kobj\_type：

```
struct kobj_type {
    void (*release)(struct kobject *kobj);
    const struct sysfs_ops *sysfs_ops;
    const struct attribute_group **default_groups;
    const struct kobj_ns_type_operations *(*child_ns_type)(struct kobject *kobj);
    const void *(*namespace)(struct kobject *kobj);
    void (*get_ownership)(struct kobject *kobj, kuid_t *uid, kgid_t *gid);
};
```

这个结构提用来描述一个特定类型的 kobject（或者更正确地说，包含对象的类型）。每个 kobject 都需要有一个相关的 kobj\_type 结构；当你调用 kobject\_init() 或 kobject\_init\_and\_add() 时必须指定一个指向该结构的指针。

当然，kobj\_type 结构中的 release 字段是指向这种类型的 kobject 的 release() 方法的一个指针。另外两

个字段 (`sysfs_ops` 和 `default_groups`) 控制这种类型的对象如何在 `sysfs` 中被表示；它们超出了本文的范围。

`default_groups` 指针是一个默认属性的列表，它将为任何用这个 `ktype` 注册的 `kobject` 自动创建。

### ksets

一个 `kset` 仅仅是一个希望相互关联的 `kobjects` 的集合。没有限制它们必须是相同的 `ktype`，但是如果它们不是相同的，就要非常小心。

一个 `kset` 有以下功能：

- 它像是一个包含一组对象的袋子。一个 `kset` 可以被内核用来追踪“所有块设备”或“所有 PCI 设备驱动”。
- `kset` 也是 `sysfs` 中的一个子目录，与 `kset` 相关的 `kobjects` 可以在这里显示出来。每个 `kset` 都包含一个 `kobject`，它可以被设置为其他 `kobject` 的父对象；`sysfs` 层次结构的顶级目录就是以这种方式构建的。
- `Ksets` 可以支持 `kobjects` 的“热插拔”，并影响 `uevent` 事件如何被报告给用户空间。

在面向对象的术语中，“`kset`”是顶级的容器类；`ksets` 包含它们自己的 `kobject`，但是这个 `kobject` 是由 `kset` 代码管理的，不应该被任何其他用户所操纵。

`kset` 在一个标准的内核链表中保存它的子对象。`Kobjects` 通过其 `kset` 字段指向其包含的 `kset`。在几乎所有的情况下，属于一个 `kset` 的 `kobjects` 在它们的父对象中都有那个 `kset`（或者，严格地说，它的嵌入 `kobject`）。

由于 `kset` 中包含一个 `kobject`，它应该总是被动态地创建，而不是静态地或在堆栈中声明。要创建一个新的 `kset`，请使用：

```
struct kset *kset_create_and_add(const char *name,
                                 const struct kset_uevent_ops *uevent_ops,
                                 struct kobject *parent_kobj);
```

当你完成对 `kset` 的处理后，调用：

```
void kset_unregister(struct kset *k);
```

来销毁它。这将从 `sysfs` 中删除该 `kset` 并递减其引用计数值。当引用计数为零时，该 `kset` 将被释放。因为对该 `kset` 的其他引用可能仍然存在，释放可能发生在 `kset_unregister()` 返回之后。

一个使用 `kset` 的例子可以在内核树中的 `samples/kobject/kset-example.c` 文件中看到。

如果一个 `kset` 希望控制与它相关的 `kobjects` 的 `uevent` 操作，它可以使用结构体 `kset_uevent_ops` 来处理它：

```
struct kset_uevent_ops {
    int (*const filter)(struct kobject *kobj);
    const char *(*const name)(struct kobject *kobj);
```

```
int (* const uevent)(struct kobject *kobj, struct kobj_uevent_env *env);
};
```

过滤器函数允许 kset 阻止一个特定 kobject 的 uevent 被发送到用户空间。如果该函数返回 0，该 uevent 将不会被发射出去。

name 函数将被调用以覆盖 uevent 发送到用户空间的 kset 的默认名称。默认情况下，该名称将与 kset 本身相同，但这个函数，如果存在，可以覆盖该名称。

当 uevent 即将被发送至用户空间时，uevent 函数将被调用，以允许更多的环境变量被添加到 uevent 中。

有人可能会问，鉴于没有提出执行该功能的函数，究竟如何将一个 kobject 添加到一个 kset 中。答案是这个任务是由 kobject\_add() 处理的。当一个 kobject 被传递给 kobject\_add() 时，它的 kset 成员应该指向这个 kobject 所属的 kset。kobject\_add() 将处理剩下的部分。

如果属于一个 kset 的 kobject 没有父 kobject 集，它将被添加到 kset 的目录中。并非所有的 kset 成员都必须住在 kset 目录中。如果在添加 kobject 之前分配了一个明确的父 kobject，那么该 kobject 将被注册到 kset 中，但是被添加到父 kobject 下面。

## 移除 Kobject

当一个 kobject 在 kobject 核心注册成功后，在代码使用完它时，必须将其清理掉。要做到这一点，请调用 kobject\_put()。通过这样做，kobject 核心会自动清理这个 kobject 分配的所有内存。如果为这个对象发送了 KOBJ\_ADD uevent，那么相应的 KOBJ\_REMOVE uevent 也将被发送，任何其他的 sysfs 内务将被正确处理。

如果你需要分两次对 kobject 进行删除（比如说在你要销毁对象时无权睡眠），那么调用 kobject\_del() 将从 sysfs 中取消 kobject 的注册。这使得 kobject “不可见”，但它并没有被清理掉，而且该对象的引用计数仍然是一样的。在稍后的时间调用 kobject\_put() 来完成与该 kobject 相关的内存的清理。

kobject\_del() 可以用来放弃对父对象的引用，如果循环引用被构建的话。在某些情况下，一个父对象引用一个子对象是有效的。循环引用必须通过明确调用 kobject\_del() 来打断，这样一个释放函数就会被调用，前一个循环中的对象会相互释放。

## 示例代码出处

关于正确使用 ksets 和 kobjects 的更完整的例子，请参见示例程序 samples/kobject/{kobject-example.c, kset-example.c}，如果您选择 CONFIG\_SAMPLE\_KOBJECT，它们将被构建为可加载模块。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的

帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

### Original Documentation/core-api/kref.rst

翻译:

司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译:

< 此处请校译员签名（自愿），我将在下一个版本添加 >

### 为内核对象添加引用计数器 (krefs)

作者 Corey Minyard <[minyard@acm.org](mailto:minyard@acm.org)>

作者 Thomas Hellstrom <[thellstrom@vmware.com](mailto:thellstrom@vmware.com)>

其中很多内容都是从 Greg Kroah-Hartman 2004 年关于 krefs 的 OLS 论文和演讲中摘录的，可以在以下网址找到：

- [http://www.kroah.com/linux/talks/ols\\_2004\\_kref\\_paper/Reprint-Kroah-Hartman-OLS2004.pdf](http://www.kroah.com/linux/talks/ols_2004_kref_paper/Reprint-Kroah-Hartman-OLS2004.pdf)
- [http://www.kroah.com/linux/talks/ols\\_2004\\_kref\\_talk/](http://www.kroah.com/linux/talks/ols_2004_kref_talk/)

### 简介

krefs 允许你为你的对象添加引用计数器。如果你有在多个地方使用和传递的对象，而你没有 refcounts，你的代码几乎肯定是坏的。如果你想要引用计数，krefs 是个好办法。

要使用 kref，请在你的数据结构中添加一个，如：

```
struct my_data
{
    .
    .
    struct kref refcount;
    .
};

};
```

kref 可以出现在数据结构体中的任何地方。

## 初始化

你必须在分配 kref 之后初始化它。要做到这一点，可以这样调用 kref\_init:

```
struct my_data *data;

data = kmalloc(sizeof(*data), GFP_KERNEL);
if (!data)
    return -ENOMEM;
kref_init(&data->refcount);
```

这将 kref 中的 refcount 设置为 1。

## Kref 规则

一旦你有一个初始化的 kref，你必须遵循以下规则：

- 1) 如果你对一个指针做了一个非临时性的拷贝，特别是如果它可以被传递给另一个执行线程，你必须在传递之前用 kref\_get() 增加 refcount:

```
kref_get(&data->refcount);
```

如果你已经有了一个指向 kref-ed 结构体的有效指针 (refcount 不能为零)，你可以在没有锁的情况下这样做。

- 2) 当你完成对一个指针的处理时，你必须调用 kref\_put():

```
kref_put(&data->refcount, data_release);
```

如果这是对该指针的最后一次引用，释放程序将被调用。如果代码从来没有尝试过在没有已经持有有效指针的情况下获得一个 kref-ed 结构体的有效指针，那么在没有锁的情况下这样做是安全的。

- 3) 如果代码试图获得对一个 kref-ed 结构体的引用，而不持有一个有效的指针，它必须按顺序访问，在 kref\_put() 期间不能发生 kref\_get()，并且该结构体在 kref\_get() 期间必须保持有效。

例如，如果你分配了一些数据，然后将其传递给另一个线程来处理：

```
void data_release(struct kref *ref)
{
    struct my_data *data = container_of(ref, struct my_data, refcount);
    kfree(data);
}

void more_data_handling(void *cb_data)
{
    struct my_data *data = cb_data;
    .
    . do stuff with data here
    .
}
```

```

        kref_put(&data->refcount, data_release);
    }

int my_data_handler(void)
{
    int rv = 0;
    struct my_data *data;
    struct task_struct *task;
    data = kmalloc(sizeof(*data), GFP_KERNEL);
    if (!data)
        return -ENOMEM;
    kref_init(&data->refcount);

    kref_get(&data->refcount);
    task = kthread_run(more_data_handling, data, "more_data_handling");
    if (task == ERR_PTR(-ENOMEM)) {
        rv = -ENOMEM;
        kref_put(&data->refcount, data_release);
        goto out;
    }

    .
    . do stuff with data here
    .

out:
    kref_put(&data->refcount, data_release);
    return rv;
}

```

这样，两个线程处理数据的顺序并不重要，`kref_put()` 处理知道数据不再被引用并释放它。`kref_get()` 不需要锁，因为我们已经有了一个有效的指针，我们拥有一个 `refcount`。`put` 不需要锁，因为没有任何东西试图在没有持有指针的情况下获取数据。

在上面的例子中，`kref_put()` 在成功和错误路径中都会被调用 2 次。这是必要的，因为引用计数被 `kref_init()` 和 `kref_get()` 递增了 2 次。

请注意，规则 1 中的“before”是非常重要的。你不应该做类似于：

```

task = kthread_run(more_data_handling, data, "more_data_handling");
if (task == ERR_PTR(-ENOMEM)) {
    rv = -ENOMEM;
    goto out;
} else
    /* BAD BAD BAD - 在交接后得到 */
    kref_get(&data->refcount);

```

不要以为你知道自己在做什么而使用上述构造。首先，你可能不知道自己在做什么。其次，你可能知道自己在做什么（有些情况下涉及到锁，上述做法可能是合法的），但其他知道自己在做什么的人可能会改变代码或复制代码。这是很危险的作风。请不要这样做。

在有些情况下，你可以优化 get 和 put。例如，如果你已经完成了一个对象，并且给其他对象排队，或者把它传递给其他对象，那么就没有理由先做一个 get，然后再做一个 put：

```
/* 糟糕的额外获取 (get) 和输出 (put) */
kref_get(&obj->ref);
enqueue(obj);
kref_put(&obj->ref, obj_cleanup);
```

只要做 enqueue 就可以了。我们随时欢迎对这个问题的评论：

```
enqueue(obj);
/* 我们已经完成了对 obj 的处理，所以我们把我们的 refcount 传给了队列。
在这之后不要再碰 obj 了！ */
```

最后一条规则（规则 3）是最难处理的一条。例如，你有一个每个项目都被 krefed 的列表，而你希望得到第一个项目。你不能只是从列表中抽出第一个项目，然后 kref\_get() 它。这违反了规则 3，因为你还没有持有一个有效的指针。你必须添加一个 mutex（或其他锁）。比如说：

```
static DEFINE_MUTEX(mutex);
static LIST_HEAD(q);
struct my_data
{
    struct kref      refcount;
    struct list_head link;
};

static struct my_data *get_entry()
{
    struct my_data *entry = NULL;
    mutex_lock(&mutex);
    if (!list_empty(&q)) {
        entry = container_of(q.next, struct my_data, link);
        kref_get(&entry->refcount);
    }
    mutex_unlock(&mutex);
    return entry;
}

static void release_entry(struct kref *ref)
{
    struct my_data *entry = container_of(ref, struct my_data, refcount);

    list_del(&entry->link);
    kfree(entry);
}

static void put_entry(struct my_data *entry)
{
    mutex_lock(&mutex);
    kref_put(&entry->refcount, release_entry);
```

```
    mutex_unlock(&mutex);
}
```

如果你不想在整个释放操作过程中持有锁，`kref_put()` 的返回值是有用的。假设你不想在上面的例子中在持有锁的情况下调用 `kfree()`（因为这样做有点无意义）。你可以使用 `kref_put()`，如下所示：

```
static void release_entry(struct kref *ref)
{
    /* 所有的工作都是在从 kref_put() 返回后完成的。 */
}

static void put_entry(struct my_data *entry)
{
    mutex_lock(&mutex);
    if (kref_put(&entry->refcount, release_entry)) {
        list_del(&entry->link);
        mutex_unlock(&mutex);
        kfree(entry);
    } else
        mutex_unlock(&mutex);
}
```

如果你必须调用其他程序作为释放操作的一部分，而这些程序可能需要很长的时间，或者可能要求相同的锁，那么这真的更有用。请注意，在释放例程中做所有的事情还是比较好的，因为它比较整洁。

上面的例子也可以用 `kref_get_unless_zero()` 来优化，方法如下：

```
static struct my_data *get_entry()
{
    struct my_data *entry = NULL;
    mutex_lock(&mutex);
    if (!list_empty(&q)) {
        entry = container_of(q.next, struct my_data, link);
        if (!kref_get_unless_zero(&entry->refcount))
            entry = NULL;
    }
    mutex_unlock(&mutex);
    return entry;
}

static void release_entry(struct kref *ref)
{
    struct my_data *entry = container_of(ref, struct my_data, refcount);

    mutex_lock(&mutex);
    list_del(&entry->link);
    mutex_unlock(&mutex);
    kfree(entry);
}
```

```
static void put_entry(struct my_data *entry)
{
    kref_put(&entry->refcount, release_entry);
}
```

这对于在 `put_entry()` 中移除 `kref_put()` 周围的 `mutex` 锁是很有用的，但是重要的是 `kref_get_unless_zero` 被封装在查找表中的同一关键部分，否则 `kref_get_unless_zero` 可能引用已经释放的内存。注意，在不检查其返回值的情况下使用 `kref_get_unless_zero` 是非法的。如果你确信（已经有了一个有效的指针）`kref_get_unless_zero()` 会返回 `true`，那么就用 `kref_get()` 代替。

## Krefs 和 RCU

函数 `kref_get_unless_zero` 也使得在上述例子中使用 `rcu` 锁进行查找成为可能：

```
struct my_data
{
    struct rcu_head rhead;
    .
    struct kref refcount;
    .
};

static struct my_data *get_entry_rcu()
{
    struct my_data *entry = NULL;
    rcu_read_lock();
    if (!list_empty(&q)) {
        entry = container_of(q.next, struct my_data, link);
        if (!kref_get_unless_zero(&entry->refcount))
            entry = NULL;
    }
    rcu_read_unlock();
    return entry;
}

static void release_entry_rcu(struct kref *ref)
{
    struct my_data *entry = container_of(ref, struct my_data, refcount);

    mutex_lock(&mutex);
    list_del_rcu(&entry->link);
    mutex_unlock(&mutex);
    kfree_rcu(entry, rhead);
}

static void put_entry(struct my_data *entry)
{
```

```
kref_put(&entry->refcount, release_entry_rcu);  
}
```

但要注意的是，在调用 `release_entry_rcu` 后，结构 `kref` 成员需要在有效内存中保留一个 `rcu` 宽限期。这可以通过使用上面的 `kfree_rcu(entry, rhead)` 来实现，或者在使用 `kfree` 之前调用 `synchronize_rcu()`，但注意 `synchronize_rcu()` 可能会睡眠相当长的时间。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:<alexs@kernel.org>)。

**Original** Documentation/core-api/assoc\_array.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译

## 通用关联数组的实现

### 简介

这个关联数组的实现是一个具有以下属性的对象容器：

1. 对象是不透明的指针。该实现不关心它们指向哪里（如果有的话）或它们指向什么（如果有的话）。

**Note:** 指向对象的指针 必须在最小有效位为零。

2. 对象不需要包含供数组使用的链接块。这允许一个对象同时位于多个数组中。相反，数组是由指向对象的元数据块组成的。
3. 对象需要索引键来定位它们在阵列中的位置。
4. 索引键必须是唯一的。插入一个与已经在数组中的且具有相同键值的对象将取代旧的对象。
5. 索引键可以是任何长度，也可以是不同的长度。
6. 索引键应该在早期就对长度进行编码，即在任何由于长度引起的变化出现之前。
7. 索引键可以包括一个哈希值，以便将对象分散到整个数组中。
8. 该数组可以迭代。对象不一定会按索引键的顺序出现。

9. 数组可以在被修改的时候进行迭代，只要 RCU 的读锁被迭代器持有。然而，请注意，在这种情况下，一些对象可能会被看到不止一次。如果这是个问题，迭代器应该锁定以防止修改。然而，除非删除，否则对象不会被错过。

10. 数组中的对象可以通过其索引键进行查询。

11. 当数组被修改时，对象可以被查询，前提是进行查询的线程持有 RCU 的读锁。

该实现在内部使用了一棵由 16 个指针节点组成的树，这些节点在每一层都由索引键的小数点进行索引，其方式与基数树相同。为了提高内存效率，可以放置快捷键，以跳过本来是一系列单占节点的地方。此外，节点将叶子对象指针打包到节点的空闲空间中，而不是做一个额外的分支，直到有对象需要被添加到一个完整的节点中。

## 公用 API

公用 API 可以在 `<linux/assoc_array.h>` 中找到。关联数组的根是以下结构：

```
struct assoc_array {
    ...
};
```

该代码是通过启用 `CONFIG_ASSOCIATIVE_ARRAY` 来选择的，以：

```
./script/config -e ASSOCIATIVE_ARRAY
```

## 编辑脚本

插入和删除功能会产生一个“编辑脚本”，以后可以应用这个脚本来实现更改，而不会造成 ENOMEM 风险。这保留了将被安装在内部树中的预分配的元数据块，并跟踪应用脚本时将从树中删除的元数据块。

在脚本应用后，这也被用来跟踪死块和死对象，以便以后可以释放它们。释放是在 RCU 宽限期过后进行的—因此允许访问功能在 RCU 读锁下进行。

脚本在 API 之外显示为一个类型为：

```
struct assoc_array_edit;
```

有两个处理脚本的功能：

1. 应用一个编辑脚本：

```
void assoc_array_apply_edit(struct assoc_array_edit *edit);
```

这将执行编辑功能，插值各种写屏障，以允许在 RCU 读锁下的访问继续进行。然后，编辑脚本将被传递给 `call_rcu()`，以释放它和它所指向的任何死的东西。

2. Cancel an edit script:

```
void assoc_array_cancel_edit(struct assoc_array_edit *edit);
```

这将立即释放编辑脚本和所有预分配的内存。如果这是为了插入，新的对象不会被这个函数释放，而是必须由调用者释放。

这些函数保证不会失败。

### 操作表

各种功能采用了一个操作表：

```
struct assoc_array_ops {  
    ...  
};
```

这指出了一些方法，所有这些方法都需要提供：

1. 从调用者数据中获取索引键的一个块：

```
unsigned long (*get_key_chunk)(const void *index_key, int level);
```

这应该返回一个由调用者提供的索引键的块，从 level 参数给出的 比特位置开始。level 参数将是 ASSOC\_ARRAY\_KEY\_CHUNK\_SIZE 的倍数，该函数应返回 ASSOC\_ARRAY\_KEY\_CHUNK\_SIZE 位。不可能出现错误。

2. 获取一个对象的索引键的一个块：

```
unsigned long (*get_object_key_chunk)(const void *object, int level);
```

和前面的函数一样，但是从数组中的一个对象而不是从调用者提供的索引键中获取数据。

3. 看看这是否是我们要找的对象：

```
bool (*compare_object)(const void *object, const void *index_key);
```

将对象与一个索引键进行比较，如果匹配则返回 true，不匹配则返回 false。

4. 对两个对象的索引键进行比较：

```
int (*diff_objects)(const void *object, const void *index_key);
```

返回指定对象的索引键与给定索引键不同的比特位置，如果它们相同，则返回-1。

5. 释放一个对象：

```
void (*free_object)(void *object);
```

释放指定的对象。注意，这可能是在调用 `assoc_array_apply_edit()` 后的一个 RCU 宽限期内调用的，所以在模块卸载时可能需要 `synchronize_rcu()`。

## 操控函数

有一些函数用于操控关联数组：

1. 初始化一个关联数组：

```
void assoc_array_init(struct assoc_array *array);
```

这将初始化一个关联数组的基础结构。它不会失败。

2. 在一个关联数组中插入/替换一个对象：

```
struct assoc_array_edit *
assoc_array_insert(struct assoc_array *array,
                  const struct assoc_array_ops *ops,
                  const void *index_key,
                  void *object);
```

这将把给定的对象插入数组中。注意，指针的最小有效位必须是 0，因为它被用来在内部标记指针的类型。

如果该键已经存在一个对象，那么它将被新的对象所取代，旧的对象将被自动释放。

`index_key` 参数应持有索引键信息，并在调用 OPP 表中的方法时传递给它们。

这个函数不对数组本身做任何改动，而是返回一个必须应用的编辑脚本。如果出现内存不足的错误，会返回 `-ENOMEM`。

调用者应专门锁定数组的其他修改器。

3. 从一个关联数组中删除一个对象：

```
struct assoc_array_edit *
assoc_array_delete(struct assoc_array *array,
                  const struct assoc_array_ops *ops,
                  const void *index_key);
```

这将从数组中删除一个符合指定数据的对象。

`index_key` 参数应持有索引键信息，并在调用 OPP 表中的方法时传递给它们。

这个函数不对数组本身做任何改动，而是返回一个必须应用的编辑脚本。`-ENOMEM` 在出现内存不足的错误时返回。如果在数组中没有找到指定的对象，将返回 `NULL`。

调用者应该对数组的其他修改者进行专门锁定。

4. 从一个关联数组中删除所有对象：

```
struct assoc_array_edit *
assoc_array_clear(struct assoc_array *array,
                  const struct assoc_array_ops *ops);
```

这个函数删除了一个关联数组中的所有对象，使其完全为空。

这个函数没有对数组本身做任何改动，而是返回一个必须应用的编辑脚本。如果出现内存不足的错误，则返回 -ENOMEM。

调用者应专门锁定数组的其他修改者。

### 5. 销毁一个关联数组，删除所有对象：

```
void assoc_array_destroy(struct assoc_array *array,
                         const struct assoc_array_ops *ops);
```

这将破坏关联数组的内容，使其完全为空。在这个函数销毁数组的同时，不允许另一个线程在 RCU 读锁下遍历数组，因为在内存释放时不执行 RCU 延迟，这需要分配内存。

调用者应该专门针对数组的其他修改者和访问者进行锁定。

### 6. 垃圾回收一个关联数组：

```
int assoc_array_gc(struct assoc_array *array,
                   const struct assoc_array_ops *ops,
                   bool (*iterator)(void *object, void *iterator_data),
                   void *iterator_data);
```

这是对一个关联数组中的对象进行迭代，并将每个对象传递给 iterator()。如果 iterator() 返回 true，该对象被保留。如果它返回 false，该对象将被释放。如果 iterator() 函数返回 true，它必须在返回之前对该对象进行适当的 refcount 递增。

如果可能的话，内部树将被打包下来，作为迭代的一部分，以减少其中的节点数量。

iterator\_data 被直接传递给 iterator()，否则会被函数忽略。

如果成功，该函数将返回 0，如果没有足够的内存，则返回 -ENOMEM。

在这个函数执行过程中，其他线程有可能在 RCU 读锁下迭代或搜索阵列。调用者应该专门针对数组的其他修改者进行锁定。

## 访问函数

有两个函数用于访问一个关联数组：

### 1. 遍历一个关联数组中的所有对象：

```
int assoc_array_iterate(const struct assoc_array *array,
                        int (*iterator)(const void *object,
                                        void *iterator_data),
                        void *iterator_data);
```

这将数组中的每个对象传递给迭代器回调函数。iterator\_data 是该函数的私有数据。

在数组被修改的同时，可以在数组上使用这个方法，前提是 RCU 读锁被持有。在这种情况下，迭代函数有可能两次看到某些对象。如果这是个问题，那么修改应该被锁定。然而，迭代算法不应该错过任何对象。

如果数组中没有对象，该函数将返回 0，否则将返回最后一次调用的迭代器函数的结果。如果对迭代函数的任何调用导致非零返回，迭代立即停止。

2. 在一个关联数组中寻找一个对象：

```
void *assoc_array_find(const struct assoc_array *array,
                      const struct assoc_array_ops *ops,
                      const void *index_key);
```

这将直接穿过数组的内部树，到达索引键所指定的对象。

这个函数可以在修改数组的同时用在数组上，前提是 RCU 读锁被持有。

如果找到对象，该函数将返回对象（并将 `*_type` 设置为对象的类型），如果没有找到对象，将返回 `NULL`。

## 索引键形式

索引键可以是任何形式的，但是由于算法没有被告知键有多长，所以强烈建议在任何由于长度而产生的变化对比较产生影响之前，索引键应该很早就包括其长度。

这将导致具有不同长度键的叶子相互分散，而具有相同长度键的叶子则聚集在一起。

我们还建议索引键以键的其余部分的哈希值开始，以最大限度地提高整个键空间的散布情况。

分散性越好，内部树就越宽，越低。

分散性差并不是一个太大的问题，因为有快捷键，节点可以包含叶子和元数据指针的混合物。

索引键是以机器字的块状来读取的。每个块被细分为每层一个 nibble (4 比特)，所以在 32 位 CPU 上这适合 8 层，在 64 位 CPU 上适合 16 层。除非散布情况真的很差，否则不太可能有超过一个字的任何特定索引键需要被使用。

## 内部工作机制

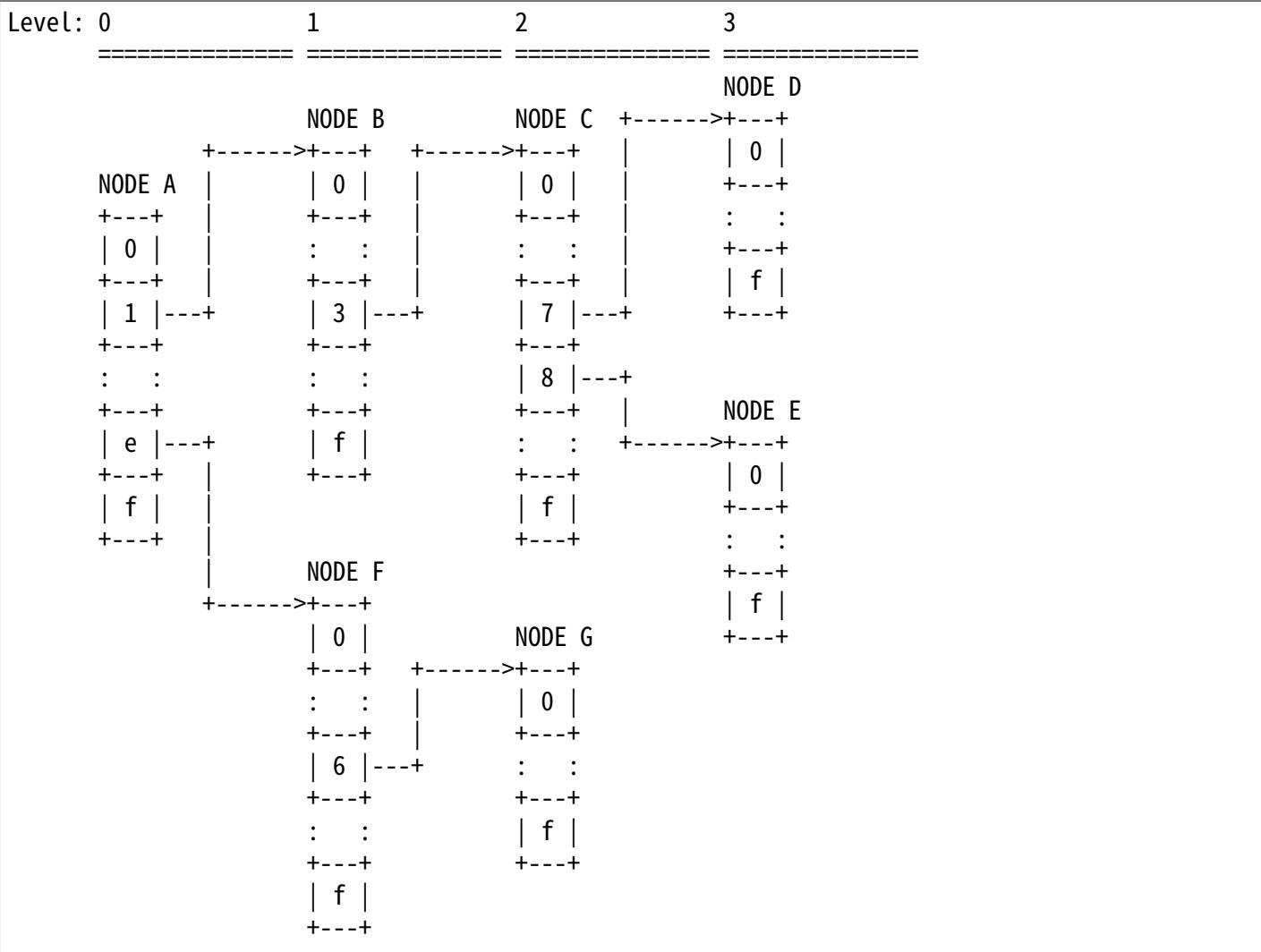
关联数组数据结构有一个内部树。这个树由两种类型的元数据块构成：节点和快捷键。

一个节点是一个槽的数组。每个槽可以包含以下四种东西之一：

- 一个 `NULL` 的指针，表示槽是空的。
- 一个指向对象（叶子）的指针。
- 一个指向下一级节点的指针。
- 一个指向快捷键的指针。

## 基本的内部树形布局

暂时不考虑快捷键，节点形成一个多层次树。索引键空间被树上的节点严格细分，节点出现在固定的层次上。例如：



在上述例子中，有 7 个节点 (A-G)，每个节点有 16 个槽 (0-f)。假设树上没有其他元数据节点，那么密钥空间是这样划分的：

KEY PREFIX	NODE
=====	====
137*	D
138*	E
13[0-69-f]*	C
1[0-24-f]*	B
e6*	G
e[0-57-f]*	F
[02-df]*	A

因此，例如，具有以下示例索引键的键将在适当的节点中被找到：

INDEX KEY	PREFIX	NODE
13694892892489	13	C
13795289025897	137	D
13889dde88793	138	E
138bbb89003093	138	E
1394879524789	12	C
1458952489	1	B
9431809de993ba	-	A
b4542910809cd	-	A
e5284310def98	e	F
e68428974237	e6	G
e7fffcbd443	e	F
f3842239082	-	A

为了节省内存，如果一个节点可以容纳它的那部分键空间中的所有叶子，那么这个节点将有所有这些叶子，而不会有元数据指针——即使其中一些叶子想在同一个槽中。

一个节点可以包含叶子和元数据指针的异质性混合。元数据指针必须在与它们的关键空间的细分相匹配的槽中。叶子可以在任何没有被元数据指针占据的槽中。保证一个节点中没有一个叶子会与元数据指针占据的槽相匹配。如果元数据指针在那里，任何键与元数据键前缀相匹配的叶必须在元数据指针指向的子树中。

在上面的索引键的例子列表中，节点 A 将包含：

SLOT	CONTENT	INDEX KEY (PREFIX)
1	PTR TO NODE B	1*
any	LEAF	9431809de993ba
any	LEAF	b4542910809cd
e	PTR TO NODE F	e*
any	LEAF	f3842239082

和节点 B：

3	PTR TO NODE C	13*
any	LEAF	1458952489

## 快捷键

快捷键是跳过一块键空间的元数据记录。快捷键是一系列通过层次上升的单占节点的替代物。快捷键的存在是为了节省内存和加快遍历速度。

树的根部有可能是一个快捷键——比如说，树至少包含 17 个节点，都有键前缀 1111。插入算法将插入一个快捷键，以单次跳过 1111 的键位，并到达第四层，在这里，这些键位实际上变得不同。

### 拆分和合并节点

每个节点的最大容量为 16 个叶子和元数据指针。如果插入算法发现它正试图将一个第 17 个对象插入到一个节点中，该节点将被拆分，使得至少两个在该层有一个共同的关键段的叶子最终在一个单独的节点中，该共同的关键段的根在该槽上。

如果一个完整的节点中的叶子和被插入的叶子足够相似，那么就会在树中插入一个快捷键。

当根植于某个节点的子树中的对象数量下降到 16 个或更少时，那么该子树将被合并成一个单独的节点——如果可能的话，这将向根部扩散。

### 非递归式迭代

每个节点和快捷键都包含一个指向其父节点的后置指针，以及该父节点中指向它的槽数。非递归迭代使用这些来通过树的根部进行，前往父节点，槽  $N+1$ ，以确保在没有堆栈的情况下取得进展。

然而，反向指针使得同时改变和迭代变得很棘手。

### 同时改变和迭代

有一些情况需要考虑：

1. 简单的插入/替换。这涉及到简单地将一个 NULL 或旧的匹配叶子的指针替换为屏障后的新叶子的指针。否则元数据块不会改变。一个旧的叶子直到 RCU 宽限期过后才会被释放。
2. 简单删除。这只是涉及到清除一个旧的匹配叶子。元数据块不会有其他变化。旧的叶子直到 RCU 宽限期之后才会被释放。
3. 插入，替换我们还没有进入的子树的一部分。这可能涉及到替换该子树的一部分——但这不会影响迭代，因为我们还没有到达它的指针，而且祖先块也不会被替换（这些块的布局不会改变）。
4. 插入替换了我们正在处理的节点。这不是一个问题，因为我们已经通过了锚定指针，直到我们跟随后面的指针才会切换到新的布局上——这时我们已经检查了被替换节点的叶子（在跟随任何元数据指针之前，我们会迭代一个节点的所有叶子）。

然而，我们可能会重新看到一些叶子，这些叶子已经被分割成一个新的分支，而这个分支的位置比我们之前的位置更远。

5. 插入替换了我们正在处理的依赖分支的节点。这不会影响到我们，直到我们跟随后面的指针。与 (4) 类似。
6. 删掉我们下面的一个分支。这不会影响我们，因为在我们看到新节点之前，回溯指针会让我们回到新节点的父节点。整个崩溃的子树被扔掉了，没有任何变化——而且仍然会在同一个槽上生根，所以我们不应该第二次处理它，因为我们会回到槽 +1。

---

**Note:** 在某些情况下，我们需要同时改变一个节点的父指针和父槽指针（比如说，我们在它之前插入了另一个节点，并把它往上移了一层）。我们不能在不锁定读取的情况下这样做——所以我们必须同时替换该节点。

然而，当我们把一个快捷键改成一个节点时，这不是一个问题，因为快捷键只有一个槽，所以当向后遍历一个槽时，不会使用父槽号。这意味着先改变槽位号是可以的——只要使用适当的屏障来确保父槽位号在后退指针之后被读取。

过时的块和叶子在 RCU 宽限期过后会被释放，所以只要任何进行遍历或迭代的人持有 RCU 读锁，旧的上层建筑就不应该在他们身上消失。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

**Original** Documentation/core-api/xarray.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译

## XArray

作者 Matthew Wilcox

### 概览

XArray 是一个抽象的数据类型，它的行为就像一个非常大的指针数组。它满足了许多与哈希或传统可调整大小的数组相同的需求。与哈希不同的是，它允许你以一种高效的缓存方式合理地转到下一个或上一个条目。与可调整大小的数组相比，不需要复制数据或改变 MMU 的映射来增加数组。与双链表相比，它的内存效率更高，可并行，对缓存更友好。它利用 RCU 的优势来执行查找而不需要锁定。

当使用的索引是密集聚集的时候，XArray 的实现是有效的；而哈希对象并使用哈希作为索引将不会有好的表现。XArray 对小的索引进行了优化，不过对大的索引仍有良好的性能。如果你的索引可以大于 `ULONG_MAX`，那么 XArray 就不适合你的数据类型。XArray 最重要的用户是页面高速缓存。

普通指针可以直接存储在 XArray 中。它们必须是 4 字节对齐的，这对任何从 `kmalloc()` 和 `alloc_page()` 返回的指针来说都是如此。这对任意的用户空间指针和函数指针来说都不是真的。你可以存储指向静态分配对象的指针，只要这些对象的对齐方式至少是 4 (字节)。

你也可以在 XArray 中存储 0 到 `LONG_MAX` 之间的整数。你必须首先使用 `xa_mk_value()` 将其转换为一个条目。当你从 XArray 中检索一个条目时，你可以通过调用 `xa_is_value()` 检查它是否是一个值条目，并通过调用 `xa_to_value()` 将它转换回一个整数。

一些用户希望对他们存储在 XArray 中的指针进行标记。你可以调用 `xa_tag_pointer()` 来创建一个带有标签的条目，`xa_untag_pointer()` 将一个有标签的条目转回一个无标签的指针，`xa_pointer_tag()` 来检索一个条目的标签。标签指针使用相同的位，用于区分值条目和普通指针，所以你必须决定他们是否要在任何特定的 XArray 中存储值条目或标签指针。

XArray 不支持存储 `IS_ERR()` 指针，因为有些指针与值条目或内部条目冲突。

XArray 的一个不寻常的特点是能够创建占据一系列索引的条目。一旦存储到其中，查询该范围内的任何索引将返回与查询该范围内任何其他索引相同的条目。存储到任何索引都会存储所有的索引条目。多索引条目可以明确地分割成更小的条目，或者将其存储 `NULL` 到任何条目中都会使 XArray 忘记范围。

## 普通 API

首先初始化一个 XArray，对于静态分配的 XArray 可以用 `DEFINE_XARRAY()`，对于动态分配的 XArray 可以用 `xa_init()`。一个新初始化的 XArray 在每个索引处都包含一个 `NULL` 指针。

然后你可以用 `xa_store()` 来设置条目，用 `xa_load()` 来获取条目。`xa_store` 将用新的条目覆盖任何条目，并返回存储在该索引的上一个条目。你可以使用 `xa_erase()` 来代替调用 `xa_store()` 的 `NULL` 条目。一个从未被存储过的条目、一个被擦除的条目和一个最近被存储过 `NULL` 的条目之间没有区别。

你可以通过使用 `xa_cmpxchg()` 有条件地替换一个索引中的条目。和 `cmpxchg()` 一样，它只有在该索引的条目有‘旧’值时才会成功。它也会返回该索引上的条目；如果它返回与传递的‘旧’相同的条目，那么 `xa_cmpxchg()` 就成功了。

如果你只想在某个索引的当前条目为 `NULL` 时将一个新条目存储到该索引，你可以使用 `xa_insert()`，如果该条目不是空的，则返回 `-EBUSY`。

你可以通过调用 `xa_extract()` 将条目从 XArray 中复制到一个普通数组中。或者你可以通过调用 `xa_for_each()`、`xa_for_each_start()` 或 `xa_for_each_range()` 来遍历 XArray 中的现有条目。你可能更喜欢使用 `xa_find()` 或 `xa_find_after()` 来移动到 XArray 中的下一个当前条目。

调用 `xa_store_range()` 可以在一个索引范围内存储同一个条目。如果你这样做，其他的一些操作将以一种稍微奇怪的方式进行。例如，在一个索引上标记条目可能会导致该条目在一些，但不是所有其他索引上被标记。储存到一个索引中可能会导致由一些，但不是所有其他索引检索的条目发生变化。

有时你需要确保对 `xa_store()` 的后续调用将不需要分配内存。`xa_reserve()` 函数将在指定索引处存储一个保留条目。普通 API 的用户将看到这个条目包含 `NULL`。如果你不需要使用保留的条目，你可以调用 `xa_release()` 来删除这个未使用的条目。如果在此期间有其他用户存储到该条目，`xa_release()` 将不做任何事情；相反，如果你想让该条目变成 `NULL`，你应该使用 `xa_erase()`。在一个保留的条目上使用 `xa_insert()` 将会失败。

如果数组中的所有条目都是 `NULL`，`xa_empty()` 函数将返回 `true`。

最后，你可以通过调用 `xa_destroy()` 删除 XArray 中的所有条目。如果 XArray 的条目是指针，你可能希望先释放这些条目。你可以通过使用 `xa_for_each()` 迭代器遍历 XArray 中所有存在的条目来实现这一目的。

## 搜索标记

数组中的每个条目都有三个与之相关的位，称为标记。每个标记可以独立于其他标记被设置或清除。你可以通过使用 `xa_for_each_marked()` 迭代器来迭代有标记的条目。

你可以通过使用 `xa_get_mark()` 来查询某个条目是否设置了标记。如果该条目不是 `NULL`，你可以通过使用 `xa_set_mark()` 来设置一个标记，并通过调用 `xa_clear_mark()` 来删除条目上的标记。你可以通过调用 `xa_marked()` 来询问 XArray 中的任何条目是否有一个特定的标记被设置。从 XArray 中删除一个条目会导致与该条目相关的所有标记被清除。

在一个多索引条目的任何索引上设置或清除标记将影响该条目所涵盖的所有索引。查询任何索引上的标记将返回相同的结果。

没有办法对没有标记的条目进行迭代；数据结构不允许有效地实现这一点。目前没有迭代器来搜索比特的逻辑组合（例如迭代所有同时设置了 `XA_MARK_1` 和 `XA_MARK_2` 的条目，或者迭代所有设置了 `XA_MARK_0` 或 `XA_MARK_2` 的条目）。如果有用户需要，可以增加这些内容。

## 分配 XArrays

如果你使用 `DEFINE_XARRAY_ALLOC()` 来定义 XArray，或者通过向 `xa_init_flags()` 传递 `XA_FLAGS_ALLOC` 来初始化它，XArray 会改变以跟踪条目是否被使用。

你可以调用 `xa_alloc()` 将条目存储在 XArray 中一个未使用的索引上。如果你需要从中断上下文中修改数组，你可以使用 `xa_alloc_bh()` 或 `xa_alloc_irq()`，在分配 ID 的同时禁用中断。

使用 `xa_store()`、`xa_cmpxchg()` 或 `xa_insert()` 也将标记该条目为正在分配。与普通的 XArray 不同，存储 `NULL` 将标记该条目为正在使用中，就像 `xa_reserve()`。要释放一个条目，请使用 `xa_erase()`（或者 `xa_release()`，如果你只想释放一个 `NULL` 的条目）。

默认情况下，最低的空闲条目从 0 开始分配。如果你想从 1 开始分配条目，使用 `DEFINE_XARRAY_ALLOC1()` 或 `XA_FLAGS_ALLOC1` 会更有效。如果你想分配 ID 到一个最大值，然后绕回最低的空闲 ID，你可以使用 `xa_alloc_cyclic()`。

你不能在分配的 XArray 中使用 `XA_MARK_0`，因为这个标记是用来跟踪一个条目是否是空闲的。其他的标记可以供你使用。

## 内存分配

`xa_store()`、`xa_cmpxchg()`、`xa_alloc()`、`xa_reserve()` 和 `xa_insert()` 函数接受一个 `gfp_t` 参数，以防 XArray 需要分配内存来存储这个条目。如果该条目被删除，则不需要进行内存分配，指定的 GFP 标志将被忽略。

没有内存可供分配是可能的，特别是如果你传递了一组限制性的 GFP 标志。在这种情况下，这些函数会返回一个特殊的值，可以用 `xa_err()` 把它变成一个错误值。如果你不需要确切地知道哪个错误发生，使用 `xa_is_err()` 会更有效一些。

### 锁

当使用普通 API 时，你不必担心锁的问题。XArray 使用 RCU 和一个内部自旋锁来同步访问：

#### 不需要锁：

- `xa_empty()`
- `xa_marked()`

#### 采取 RCU 读锁：

- `xa_load()`
- `xa_for_each()`
- `xa_for_each_start()`
- `xa_for_each_range()`
- `xa_find()`
- `xa_find_after()`
- `xa_extract()`
- `xa_get_mark()`

#### 内部使用 `xa_lock`：

- `xa_store()`
- `xa_store_bh()`
- `xa_store_irq()`
- `xa_insert()`
- `xa_insert_bh()`
- `xa_insert_irq()`
- `xa_erase()`
- `xa_erase_bh()`
- `xa_erase_irq()`
- `xa_cmpxchg()`
- `xa_cmpxchg_bh()`
- `xa_cmpxchg_irq()`
- `xa_store_range()`
- `xa_alloc()`
- `xa_alloc_bh()`

- `xa_alloc_irq()`
- `xa_reserve()`
- `xa_reserve_bh()`
- `xa_reserve_irq()`
- `xa_destroy()`
- `xa_set_mark()`
- `xa_clear_mark()`

假设进入时持有 `xa_lock`:

- `_xa_store()`
- `_xa_insert()`
- `_xa_erase()`
- `_xa_cmpxchg()`
- `_xa_alloc()`
- `_xa_set_mark()`
- `_xa_clear_mark()`

如果你想利用锁来保护你存储在 XArray 中的数据结构，你可以在调用 `xa_load()` 之前调用 `xa_lock()`，然后在调用 `xa_unlock()` 之前对你找到的对象进行一个引用计数。这将防止存储操作在查找对象和增加 refcount 期间从数组中删除对象。你也可以使用 RCU 来避免解除对已释放内存的引用，但对这一点的解释已经超出了本文的范围。

在修改数组时，XArray 不会禁用中断或 softirqs。从中断或 softirq 上下文中读取 XArray 是安全的，因为 RCU 锁提供了足够的保护。

例如，如果你想在进程上下文中存储 XArray 中的条目，然后在 softirq 上下文中擦除它们，你可以这样做:

```
void foo_init(struct foo *foo)
{
    xa_init_flags(&foo->array, XA_FLAGS_LOCK_BH);
}

int foo_store(struct foo *foo, unsigned long index, void *entry)
{
    int err;

    xa_lock_bh(&foo->array);
    err = xa_err(_xa_store(&foo->array, index, entry, GFP_KERNEL));
    if (!err)
        foo->count++;
    xa_unlock_bh(&foo->array);
    return err;
}
```

```

}

/* foo_erase() 只在软中断上下文中调用 */
void foo_erase(struct foo *foo, unsigned long index)
{
    xa_lock(&foo->array);
    __xa_erase(&foo->array, index);
    foo->count--;
    xa_unlock(&foo->array);
}

```

如果你要从中断或 softirq 上下文中修改 XArray，你需要使用 `xa_init_flags()` 初始化数组，传递 `XA_FLAGS_LOCK_IRQ` 或 `XA_FLAGS_LOCK_BH`（参数）。

上面的例子还显示了一个常见的模式，即希望在存储端扩展 `xa_lock` 的覆盖范围，以保护与数组相关的一些统计数据。

与中断上下文共享 XArray 也是可能的，可以在中断处理程序和进程上下文中都使用 `xa_lock_irqsave()`，或者在进程上下文中使用 `xa_lock_irq()`，在中断处理程序中使用 `xa_lock()`。一些更常见的模式有一些辅助函数，如 `xa_store_bh()`、`xa_store_irq()`、`xa_erase_bh()`、`xa_erase_irq()`、`xa_cmpxchg_bh()` 和 `xa_cmpxchg_irq()`。

有时你需要用一个 `mutex` 来保护对 XArray 的访问，因为这个锁在锁的层次结构中位于另一个 `mutex` 之上。这并不意味着你有权使用像 `_xa_erase()` 这样的函数而不占用 `xa_lock`；`xa_lock` 是用来进行 lockdep 验证的，将来也会用于其他用途。

`_xa_set_mark()` 和 `_xa_clear_mark()` 函数也适用于你查找一个条目并想原子化地设置或清除一个标记的情况。在这种情况下，使用高级 API 可能更有效，因为它将使你免于走两次树。

## 高级 API

高级 API 提供了更多的灵活性和更好的性能，但代价是接口可能更难使用，保障措施更少。高级 API 没有为你加锁，你需要在修改数组的时候使用 `xa_lock`。在对数组进行只读操作时，你可以选择使用 `xa_lock` 或 RCU 锁。你可以在同一个数组上混合使用高级和普通操作；事实上，普通 API 是以高级 API 的形式实现的。高级 API 只对具有 GPL 兼容许可证的模块可用。

高级 API 是基于 `xa_state` 的。这是一个不透明的数据结构，你使用 `XA_STATE()` 宏在堆栈中声明。这个宏初始化了 `xa_state`，准备开始在 XArray 上移动。它被用作一个游标来保持在 XArray 中的位置，并让你把各种操作组合在一起，而不必每次都从头开始。

`xa_state` 也被用来存储错误 (store errors)。你可以调用 `xas_error()` 来检索错误。所有的操作在进行之前都会检查 `xa_state` 是否处于错误状态，所以你没有必要在每次调用之后检查错误；你可以连续进行多次调用，只在方便的时候检查。目前 XArray 代码本身产生的错误只有 `ENOMEM` 和 `EINVAL`，但它支持任意的错误，以防你想自己调用 `xas_set_err()`。

如果 `xa_state` 持有 `ENOMEM` 错误，调用 `xas_nomem()` 将尝试使用指定的 `gfp` 标志分配更多的内存，并将其缓存在 `xa_state` 中供下一次尝试。这个想法是，你拿着 `xa_lock`，尝试操作，然后放弃锁。该操作试图在持

有锁的情况下分配内存，但它更有可能失败。一旦你放弃了锁，`xas_nomem()` 可以更努力地尝试分配更多内存。如果值得重试该操作，它将返回 `true`（即出现了内存错误，分配了更多的内存）。如果它之前已经分配了内存，并且该内存没有被使用，也没有错误（或者一些不是 `ENOMEM` 的错误），那么它将释放之前分配的内存。

## 内部条目

XArray 为它自己的目的保留了一些条目。这些条目从未通过正常的 API 暴露出来，但是当使用高级 API 时，有可能看到它们。通常，处理它们的最好方法是把它们传递给 `xas_retry()`，如果它返回 `true`，就重试操作。

名称	检测	用途
Node	<code>xa_is_node()</code>	一个 XArray 节点。在使用多索引 <code>xa_state</code> 时可能是可见的。
Sibling	<code>xa_is_sibling()</code>	一个多索引条目的非典型条目。该值表示该节点中的哪个槽有典型条目。
Retry	<code>xa_is_retry()</code>	这个条目目前正在被一个拥有 <code>xa_lock</code> 的线程修改。在这个 RCU 周期结束时，包含该条目的节点可能会被释放。你应该从数组的头部重新开始查找。
Zero	<code>xa_is_zero()</code>	Zero 条目通过普通 API 显示为 <code>NULL</code> ，但在 XArray 中占有一个条目，可用于保留索引供将来使用。这是通过为分配的条目分配 XArrays 来使用的，这些条目是 <code>NULL</code> 。

其他内部条目可能会在未来被添加。在可能的情况下，它们将由 `xas_retry()` 处理。

## 附加函数

`xas_create_range()` 函数分配了所有必要的内存来存储一个范围内的每一个条目。如果它不能分配内存，它将在 `xa_state` 中设置 `ENOMEM`。

你可以使用 `xas_init_marks()` 将一个条目上的标记重置为默认状态。这通常是清空所有标记，除非 XArray 被标记为 `XA_FLAGS_TRACK_FREE`，在这种情况下，标记 0 被设置，所有其他标记被清空。使用 `xas_store()` 将一个条目替换为另一个条目不会重置该条目上的标记；如果你想重置标记，你应该明确地这样做。

`xas_load()` 会尽可能地将 `xa_state` 移动到该条目附近。如果你知道 `xa_state` 已经移动到了该条目，并且需要检查该条目是否有变化，你可以使用 `xas_reload()` 来保存一个函数调用。

如果你需要移动到 XArray 中的不同索引，可以调用 `xas_set()`。这可以将光标重置到树的顶端，这通常会使下一个操作将光标移动到树中想要的位置。如果你想移动到下一个或上一个索引，调用 `xas_next()` 或 `xas_prev()`。设置索引不会使光标在数组中移动，所以不需要锁，而移动到下一个或上一个索引则需要锁。

你可以使用 `xas_find()` 搜索下一个当前条目。这相当于 `xa_find()` 和 `xa_find_after()`；如果光标已经移动到了一个条目，那么它将找到当前引用的条目之后的下一个条目。如果没有，它将返回 `xa_state` 索引处的条目。使用 `xas_next_entry()` 而不是 `xas_find()` 来移动到下一个当前条目，在大多数情况下会节省一个函数调用，但代价是发出更多内联代码。

`xas_find_marked()` 函数也是如此。如果 `xa_state` 没有被移动过，它将返回 `xa_state` 的索引处的条目，如果它被标记了。否则，它将返回 `xa_state` 所引用的条目之后的第一个被标记的条目。`xas_next_marked()` 函数等同于 `xas_next_entry()`。

当使用 `xas_for_each()` 或 `xas_for_each_marked()` 在 XArray 的某个范围内进行迭代时，可能需要暂时停止迭代。`xas_pause()` 函数的存在就是为了这个目的。在你完成了必要的工作并希望恢复后，`xa_state` 处于适当的状态，在你最后处理的条目后继续迭代。如果你在迭代时禁用了中断，那么暂停迭代并在每一个 `XA_CHECK_SCHED` 条目中重新启用中断是很好的做法。

`xas_get_mark()`, `xas_set_mark()` 和 `xas_clear_mark()` 函数要求 `xa_state` 光标已经被移动到 XArray 中的适当位置；如果你在之前调用了 `xas_pause()` 或 `xas_set()`，它们将不会有任何作用。

你可以调用 `xas_set_update()`，让 XArray 每次更新一个节点时都调用一个回调函数。这被页面缓存的 `workingset` 代码用来维护其只包含阴影项的节点列表。

### 多索引条目

XArray 有能力将多个索引联系在一起，因此对一个索引的操作会影响到所有的索引。例如，存储到任何索引将改变从任何索引检索的条目的值。在任何索引上设置或清除一个标记，都会在每个被绑在一起的索引上设置或清除该标记。目前的实现只允许将 2 的整数倍的范围绑在一起；例如指数 64-127 可以绑在一起，但 2-6 不能。这可以节省大量的内存；例如，将 512 个条目绑在一起可以节省 4kB 以上的内存。

你可以通过使用 `XA_STATE_ORDER()` 或 `xas_set_order()`，然后调用 `xas_store()` 来创建一个多索引条目。用一个多索引的 `xa_state` 调用 `xas_load()` 会把 `xa_state` 移动到树中的正确位置，但是返回值没有意义，有可能是一个内部条目或 `NULL`，即使在范围内有一个条目存储。调用 `xas_find_conflict()` 将返回该范围内的第一个条目，如果该范围内没有条目，则返回 `NULL`。`xas_for_each_conflict()` 迭代器将遍历每个与指定范围重叠的条目。

如果 `xas_load()` 遇到一个多索引条目，`xa_state` 中的 `xa_index` 将不会被改变。当遍历一个 XArray 或者调用 `xas_find()` 时，如果初始索引在一个多索引条目的中间，它将不会被改变。随后的调用或迭代将把索引移到范围内的第一个索引。每个条目只会被返回一次，不管它占据了多少个索引。

不支持使用 `xas_next()` 或 `xas_prev()` 来处理一个多索引的 `xa_state`。在一个多索引的条目上使用这两个函数中的任何一个都会显示出同级的条目；这些条目应该被调用者跳过。

在一个多索引条目的任何一个索引中存储 `NULL`，将把每个索引的条目设置为 `NULL`，并解除绑定。通过调用 `xas_split_alloc()`，在没有 `xa_lock` 的情况下，可以将一个多索引条目分割成占据较小范围的条目，然后再取锁并调用 `xas_split()`。

### 函数和结构体

该 API 在以下内核代码中：

include/linux/xarray.h  
lib/xarray.c

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题, 请联系该文件的译者, 或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/core-api/rbtree.rst

翻译 唐艺舟 Tang Yizhou <[tangyeechou@gmail.com](mailto:tangyeechou@gmail.com)>

## Linux 中的红黑树 (rbtree)

日期 2007 年 1 月 18 日

作者 Rob Landley <[rob@landley.net](mailto:rob@landley.net)>

### 何为红黑树, 它们有什么用?

红黑树是一种自平衡二叉搜索树, 被用来存储可排序的键/值数据对。这与基数树 (被用来高效存储稀疏数组, 因此使用长整型下标来插入/访问/删除结点) 和哈希表 (没有保持排序因而无法容易地按序遍历, 同时必须调节其大小和哈希函数, 然而红黑树可以优雅地伸缩以便存储任意数量的键) 不同。

红黑树和 AVL 树类似, 但在插入和删除时提供了更快的实时有界的最坏情况性能 (分别最多两次旋转和三次旋转, 来平衡树), 查询时间轻微变慢 (但时间复杂度仍然是  $O(\log n)$ )。

引用 Linux 每周新闻 (Linux Weekly News):

内核中有多处红黑树的使用案例。最后期限调度器和完全公平排队 (CFQ) I/O 调度器利用红黑树跟踪请求; 数据包 CD/DVD 驱动程序也是如此。高精度时钟代码使用一颗红黑树组织未完成的定时器请求。ext3 文件系统用红黑树跟踪目录项。虚拟内存区域 (VMAs)、epoll 文件描述符、密码学密钥和在“分层令牌桶”调度器中的网络数据包都由红黑树跟踪。

本文档涵盖了对 Linux 红黑树实现的使用方法。更多关于红黑树的性质和实现的信息, 参见:

**Linux 每周新闻关于红黑树的文章** <https://lwn.net/Articles/184495/>

**维基百科红黑树词条** [https://en.wikipedia.org/wiki/Red-black\\_tree](https://en.wikipedia.org/wiki/Red-black_tree)

### 红黑树的 Linux 实现

Linux 的红黑树实现在文件 “lib/rbtree.c” 中。要使用它, 需要 “`#include <linux/rbtree.h>`”。

Linux 的红黑树实现对速度进行了优化, 因此比传统的实现少一个间接层 (有更好的缓存局部性)。每个 `rb_node` 结构体的实例嵌入在它管理的数据结构中, 因此不需要靠指针来分离 `rb_node` 和它管理的数据结构。用户应该编写他们自己的树搜索和插入函数, 来调用已提供的红黑树函数, 而不是使用一个比较回调函数指针。加锁代码也留给红黑树的用户编写。

## 创建一颗红黑树

红黑树中的数据结点是包含 rb\_node 结构体成员的结构体:

```
struct mytype {
    struct rb_node node;
    char *keystring;
};
```

当处理一个指向内嵌 rb\_node 结构体的指针时，包住 rb\_node 的结构体可用标准的 container\_of() 宏访问。此外，个体成员可直接用 rb\_entry(node, type, member) 访问。

每颗红黑树的根是一个 rb\_root 数据结构，它由以下方式初始化为空:

```
struct rb_root mytree = RB_ROOT;
```

## 在一颗红黑树中搜索值

为你的树写一个搜索函数是相当简单的：从树根开始，比较每个值，然后根据需要继续前往左边或右边的分支。

示例:

```
struct mytype *my_search(struct rb_root *root, char *string)
{
    struct rb_node *node = root->rb_node;

    while (node) {
        struct mytype *data = container_of(node, struct mytype, node);
        int result;

        result = strcmp(string, data->keystring);

        if (result < 0)
            node = node->rb_left;
        else if (result > 0)
            node = node->rb_right;
        else
            return data;
    }
    return NULL;
}
```

## 在一颗红黑树中插入数据

在树中插入数据的步骤包括：首先搜索插入新结点的位置，然后插入结点并对树再平衡（“ recoloring”）。

插入的搜索和上文的搜索不同，它要找到嫁接新结点的位置。新结点也需要一个指向它的父节点的链接，以达到再平衡的目的。

示例：

```
int my_insert(struct rb_root *root, struct mytype *data)
{
    struct rb_node **new = &(root->rb_node), *parent = NULL;

    /* Figure out where to put new node */
    while (*new) {
        struct mytype *this = container_of(*new, struct mytype, node);
        int result = strcmp(data->keystring, this->keystring);

        parent = *new;
        if (result < 0)
            new = &((*new)->rb_left);
        else if (result > 0)
            new = &((*new)->rb_right);
        else
            return FALSE;
    }

    /* Add new node and rebalance tree. */
    rb_link_node(&data->node, parent, new);
    rb_insert_color(&data->node, root);

    return TRUE;
}
```

## 在一颗红黑树中删除或替换已经存在的数据

若要从树中删除一个已经存在的结点，调用：

```
void rb_erase(struct rb_node *victim, struct rb_root *tree);
```

示例：

```
struct mytype *data = mysearch(&mytree, "walrus");

if (data) {
    rb_erase(&data->node, &mytree);
    myfree(data);
}
```

若要用一个新结点替换树中一个已经存在的键值相同的结点，调用：

```
void rb_replace_node(struct rb_node *old, struct rb_node *new,
                     struct rb_root *tree);
```

通过这种方式替换结点不会对树做重排序：如果新结点的键值和旧结点不同，红黑树可能被破坏。

### (按排序的顺序) 遍历存储在红黑树中的元素

我们提供了四个函数，用于以排序的方式遍历一颗红黑树的内容。这些函数可以在任意红黑树上工作，并且不需要被修改或包装（除非加锁的目的）：

```
struct rb_node *rb_first(struct rb_root *tree);
struct rb_node *rb_last(struct rb_root *tree);
struct rb_node *rb_next(struct rb_node *node);
struct rb_node *rb_prev(struct rb_node *node);
```

要开始迭代，需要使用一个指向树根的指针调用 `rb_first()` 或 `rb_last()`，它将返回一个指向树中第一个或最后一个元素所包含的节点结构的指针。要继续的话，可以在当前结点上调用 `rb_next()` 或 `rb_prev()` 来获取下一个或上一个结点。当没有剩余的结点时，将返回 `NULL`。

迭代器函数返回一个指向被嵌入的 `rb_node` 结构体的指针，由此，包住 `rb_node` 的结构体可用标准的 `container_of()` 宏访问。此外，个体成员可直接用 `rb_entry(node, type, member)` 访问。

示例：

```
struct rb_node *node;
for (node = rb_first(&mytree); node; node = rb_next(node))
    printk("key=%s\n", rb_entry(node, struct mytype, node)->keystring);
```

### 带缓存的红黑树

计算最左边（最小的）结点是二叉搜索树的一个相当常见的任务，例如用于遍历，或用户根据他们自己的逻辑依赖一个特定的顺序。为此，用户可以使用’`struct rb_root_cached`’来优化时间复杂度为  $O(\log N)$  的 `rb_first()` 的调用，以简单地获取指针，避免了潜在的昂贵的树迭代。维护操作的额外运行时间开销可忽略，不过内存占用较大。

和 `rb_root` 结构体类似，带缓存的红黑树由以下方式初始化为空：

```
struct rb_root_cached mytree = RB_ROOT_CACHED;
```

带缓存的红黑树只是一个常规的 `rb_root`，加上一个额外的指针来缓存最左边的节点。这使得 `rb_root_cached` 可以存在于 `rb_root` 存在的任何地方，并且只需增加几个接口来支持带缓存的树：

```
struct rb_node *rb_first_cached(struct rb_root_cached *tree);
void rb_insert_color_cached(struct rb_node *, struct rb_root_cached *, bool);
void rb_erase_cached(struct rb_node *node, struct rb_root_cached *);
```

操作和删除也有对应的带缓存的树的调用:

```
void rb_insert_augmented_cached(struct rb_node *node, struct rb_root_cached *,
                                bool, struct rb_augment_callbacks *);
void rb_erase_augmented_cached(struct rb_node *, struct rb_root_cached *,
                               struct rb_augment_callbacks *);
```

## 对增强型红黑树的支持

增强型红黑树是一种在每个结点里存储了“一些”附加数据的红黑树，其中结点 N 的附加数据必须是以 N 为根的子树中所有结点的内容的函数。它是建立在红黑树基础设施之上的可选特性。想要使用这个特性的红黑树用户，插入和删除结点时必须调用增强型接口并提供增强型回调函数。

实现增强型红黑树操作的 C 文件必须包含 `<linux/rbtree_augmented.h>` 而不是 `<linux/rbtree.h>`。注意，`linux/rbtree_augmented.h` 暴露了一些红黑树实现的细节而你不应依赖它们，请坚持使用文档记录的 API，并且不要在头文件中包含 `<linux/rbtree_augmented.h>`，以最小化你的用户意外地依赖这些实现细节的可能。

插入时，用户必须更新通往被插入节点的路径上的增强信息，然后像往常一样调用 `rb_link_node()`，然后是 `rb_augment_inserted()` 而不是平时的 `rb_insert_color()` 调用。如果 `rb_augment_inserted()` 再平衡了红黑树，它将回调至一个用户提供的函数来更新受影响的子树上的增强信息。

删除一个结点时，用户必须调用 `rb_erase_augmented()` 而不是 `rb_erase()`。`rb_erase_augmented()` 回调至一个用户提供的函数来更新受影响的子树上的增强信息。

在两种情况下，回调都是通过 `rb_augment_callbacks` 结构体提供的。必须定义 3 个回调：

- 一个传播回调，它更新一个给定结点和它的祖先们的增强数据，直到一个给定的停止点（如果是 NULL，将更新一路更新到树根）。
- 一个复制回调，它将一颗给定子树的增强数据复制到一个新指定的子树树根。
- 一个树旋回调，它将一颗给定的子树的增强值复制到新指定的子树树根上，并重新计算先前的子树树根的增强值。

`rb_erase_augmented()` 编译后的代码可能会内联传播、复制回调，这将导致函数体积更大，因此每个增强型红黑树的用户应该只有一个 `rb_erase_augmented()` 的调用点，以限制编译后的代码大小。

## 使用示例

区间树是增强型红黑树的一个例子。参考 Cormen, Leiserson, Rivest 和 Stein 写的《算法导论》。区间树的更多细节：

经典的红黑树只有一个键，它不能直接用来存储像 [lo:hi] 这样的区间范围，也不能快速查找与新的 lo:hi 重叠的部分，或者查找是否有与新的 lo:hi 完全匹配的部分。

然而，红黑树可以被增强，以一种结构化的方式来存储这种区间范围，从而使高效的查找和精确匹配成为可能。

这个存储在每个节点中的“额外信息”是其所有后代结点中的最大 hi (max\_hi) 值。这个信息可以保持在每个结点上，只需查看一下该结点和它的直系子结点们。这将被用于时间复杂度为 O(log n) 的最低匹配查找(所有可能的匹配中最低的起始地址)，就像这样：

```
struct interval_tree_node *
interval_tree_first_match(struct rb_root *root,
                           unsigned long start, unsigned long last)
{
    struct interval_tree_node *node;

    if (!root->rb_node)
        return NULL;
    node = rb_entry(root->rb_node, struct interval_tree_node, rb);

    while (true) {
        if (node->rb.rb_left) {
            struct interval_tree_node *left =
                rb_entry(node->rb.rb_left,
                         struct interval_tree_node, rb);
            if (left->__subtree_last >= start) {
                /*
                 * Some nodes in left subtree satisfy Cond2.
                 * Iterate to find the leftmost such node N.
                 * If it also satisfies Cond1, that's the match
                 * we are looking for. Otherwise, there is no
                 * matching interval as nodes to the right of N
                 * can't satisfy Cond1 either.
                */
                node = left;
                continue;
            }
        }
        if (node->start <= last) /* Cond1 */
            if (node->last >= start) /* Cond2 */
                return node; /* node is leftmost match */
        if (node->rb.rb_right) {
            node = rb_entry(node->rb.rb_right,
                            struct interval_tree_node, rb);
            if (node->__subtree_last >= start)
                continue;
        }
    }
}
```

```

        }
        return NULL; /* No match */
    }
}

```

插入/删除是通过以下增强型回调来定义的:

```

static inline unsigned long
compute_subtree_last(struct interval_tree_node *node)
{
    unsigned long max = node->last, subtree_last;
    if (node->rb.rb_left) {
        subtree_last = rb_entry(node->rb.rb_left,
                               struct interval_tree_node, rb)->__subtree_last;
        if (max < subtree_last)
            max = subtree_last;
    }
    if (node->rb.rb_right) {
        subtree_last = rb_entry(node->rb.rb_right,
                               struct interval_tree_node, rb)->__subtree_last;
        if (max < subtree_last)
            max = subtree_last;
    }
    return max;
}

static void augment_propagate(struct rb_node *rb, struct rb_node *stop)
{
    while (rb != stop) {
        struct interval_tree_node *node =
            rb_entry(rb, struct interval_tree_node, rb);
        unsigned long subtree_last = compute_subtree_last(node);
        if (node->__subtree_last == subtree_last)
            break;
        node->__subtree_last = subtree_last;
        rb = rb_parent(&node->rb);
    }
}

static void augment_copy(struct rb_node *rb_old, struct rb_node *rb_new)
{
    struct interval_tree_node *old =
        rb_entry(rb_old, struct interval_tree_node, rb);
    struct interval_tree_node *new =
        rb_entry(rb_new, struct interval_tree_node, rb);

    new->__subtree_last = old->__subtree_last;
}

static void augment_rotate(struct rb_node *rb_old, struct rb_node *rb_new)

```

```
{  
    struct interval_tree_node *old =  
        rb_entry(rb_old, struct interval_tree_node, rb);  
    struct interval_tree_node *new =  
        rb_entry(rb_new, struct interval_tree_node, rb);  
  
    new->_subtree_last = old->_subtree_last;  
    old->_subtree_last = compute_subtree_last(old);  
}  
  
static const struct rb_augment_callbacks augment_callbacks = {  
    augment_propagate, augment_copy, augment_rotate  
};  
  
void interval_tree_insert(struct interval_tree_node *node,  
                         struct rb_root *root)  
{  
    struct rb_node **link = &root->rb_node, *rb_parent = NULL;  
    unsigned long start = node->start, last = node->last;  
    struct interval_tree_node *parent;  
  
    while (*link) {  
        rb_parent = *link;  
        parent = rb_entry(rb_parent, struct interval_tree_node, rb);  
        if (parent->_subtree_last < last)  
            parent->_subtree_last = last;  
        if (start < parent->start)  
            link = &parent->rb.rb_left;  
        else  
            link = &parent->rb.rb_right;  
    }  
  
    node->_subtree_last = last;  
    rb_link_node(&node->rb, rb_parent, link);  
    rb_insert_augmented(&node->rb, root, &augment_callbacks);  
}  
  
void interval_tree_remove(struct interval_tree_node *node,  
                         struct rb_root *root)  
{  
    rb_erase_augmented(&node->rb, root, &augment_callbacks);  
}
```

Todolist:

idr circular-buffers generic-radix-tree packing bus-virt-phys-mapping this\_cpu\_ops  
timekeeping errseq

## 并发原语

Linux 如何让一切同时发生。详情请参阅 [/locking/index](#)

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:<alexs@kernel.org>)。

**Original** Documentation/core-api/irq/index.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## IRQs

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:<alexs@kernel.org>)。

**Original** Documentation/core-api/irq/concepts.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## 什么是 IRQ?

IRQ (Interrupt ReQuest) 指来自设备的中断请求。目前，它们可以通过一个引脚或通过一个数据包进入。多个设备可以连接到同一个引脚，从而共享一个 IRQ。

IRQ 编号是用来描述硬件中断源的内核标识符。通常它是一个到全局 `irq_desc` 数组的索引，但是除了在 `linux/interrupt.h` 中实现的之外，其它细节是体系结构特征相关的。

IRQ 编号是对机器上可能的中断源的枚举。通常枚举的是系统中所有中断控制器的输入引脚编号。在 ISA (工业标准体系结构) 的情况下所枚举的是两个 i8259 中断控制器的 16 个输入引脚。

体系结构可以给 IRQ 号赋予额外的含义，在涉及到硬件手动配置的情况下，我们鼓励这样做。ISA IRQ 是赋予这种额外含义的一个典型例子。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

**Original** Documentation/core-api/irq/irq-affinity.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## SMP IRQ 亲和性

变更记录：

- 作者：最初由 Ingo Molnar <[mingo@redhat.com](mailto:mingo@redhat.com)> 开始撰写
- 后期更新维护：Max Krasnyansky <[maxk@qualcomm.com](mailto:maxk@qualcomm.com)>

/proc/irq/IRQ#/smp\_affinity 和/proc/irq/IRQ#/smp\_affinity\_list 指定了哪些 CPU 能够关联到一个给定的 IRQ 源，这两个文件包含了这些指定 cpu 的 cpu 位掩码 (smp\_affinity) 和 cpu 列表 (smp\_affinity\_list)。它不允许关闭所有 CPU，同时如果 IRQ 控制器不支持中断请求亲和 (IRQ affinity)，那么所有 cpu 的默认值将保持不变 (即关联到所有 CPU)。

/proc/irq/default\_smp\_affinity 指明了适用于所有非激活 IRQ 的默认亲和性掩码。一旦 IRQ 被分配/激活，它的亲和位掩码将被设置为默认掩码。然后可以如上所述改变它。默认掩码是 0xffffffff。

下面是一个先将 IRQ44(eth1) 限制在 CPU0-3 上，然后限制在 CPU4-7 上的例子 (这是一个 8CPU 的 SMP box)

```
[root@moon 44]# cd /proc/irq/44
[root@moon 44]# cat smp_affinity
ffffffff

[root@moon 44]# echo 0f > smp_affinity
[root@moon 44]# cat smp_affinity
0000000f

[root@moon 44]# ping -f h
PING hell (195.4.7.3): 56 data bytes
...
--- hell ping statistics ---
6029 packets transmitted, 6027 packets received, 0% packet loss
round-trip min/avg/max = 0.1/0.1/0.4 ms
[root@moon 44]# cat /proc/interrupts | grep 'CPU\|44:'
      CPU0      CPU1      CPU2      CPU3      CPU4      CPU5      CPU6      CPU7
44:    1068     1785     1785     1783       0       0       0       0
  ↳ I/O-APIC-level  eth1
```

从上面一行可以看出，IRQ44 只传递给前四个处理器（0-3）。现在让我们把这个 IRQ 限制在 CPU(4-7)。

```
[root@moon 44]# echo f0 > smp_affinity
[root@moon 44]# cat smp_affinity
000000f0
[root@moon 44]# ping -f h
PING hell (195.4.7.3): 56 data bytes
..
--- hell ping statistics ---
2779 packets transmitted, 2777 packets received, 0% packet loss
round-trip min/avg/max = 0.1/0.5/585.4 ms
[root@moon 44]# cat /proc/interrupts | 'CPU\|44:'
      CPU0      CPU1      CPU2      CPU3      CPU4      CPU5      CPU6      CPU7
44:    1068     1785     1785     1783     1784     1069     1070     1069
  ↳ IO-APIC-level eth1
```

这次 IRQ44 只传递给最后四个处理器。即 CPU0-3 的计数器没有变化。

下面是一个将相同的 irq(44) 限制在 cpus 1024 到 1031 的例子

```
[root@moon 44]# echo 1024-1031 > smp_affinity_list
[root@moon 44]# cat smp_affinity_list
1024-1031
```

需要注意的是，如果要用位掩码来做这件事，就需要 32 个为 0 的位掩码来追踪其相关的一个。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/core-api/irq/irq-domain.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## irq\_domain 中断号映射库

目前 Linux 内核的设计使用了一个巨大的数字空间，每个独立的 IRQ 源都被分配了一个不同的数字。当只有一个中断控制器时，这很简单，但在有多个中断控制器的系统中，内核必须确保每个中断控制器都能得到非重复的 Linux IRQ 号（数字）分配。

注册为唯一的 irqchips 的中断控制器编号呈现出上升的趋势：例如 GPIO 控制器等不同种类的子驱动程序通过将其中断处理程序建模为 irqchips，即实际上是级联中断控制器，避免了重新实现与 IRQ 核心系统相同的回调机制。

在这里，中断号与硬件中断号离散了所有种类的对应关系：而在过去，IRQ 号可以选择，使它们与硬件 IRQ 线进入根中断控制器（即实际向 CPU 发射中断线的组件）相匹配，现在这个编号仅仅是一个数字。

出于这个原因，我们需要一种机制将控制器本地中断号（即硬件 irq 编号）与 Linux IRQ 号分开。

`irq_alloc_desc*`() 和 `irq_free_desc*`() API 提供了对 irq 号的分配，但它们不提供任何对控制器本地 IRQ(hwirq) 号到 Linux IRQ 号空间的反向映射的支持。

`irq_domain` 库在 `irq_alloc_desc*`() API 的基础上增加了 hwirq 和 IRQ 号码之间的映射。相比于中断控制器驱动开放编码自己的反向映射方案，我们更喜欢用 `irq_domain` 来管理映射。

`irq_domain` 还实现了从抽象的 `irq_fwspec` 结构体到 hwirq 号的转换（到目前为止是 Device Tree 和 ACPI GSI），并且可以很容易地扩展以支持其它 IRQ 拓扑数据源。

### irq\_domain 的用法

中断控制器驱动程序通过以下方式创建并注册一个 `irq_domain`。调用 `irq_domain_add_*`() 或 `irq_domain_create_*`() 函数之一（每个映射方法都有不同的分配器函数，后面会详细介绍）。函数成功后会返回一个指向 `irq_domain` 的指针。调用者必须向分配器函数提供一个 `irq_domain_ops` 结构体。

在大多数情况下，`irq_domain` 在开始时是空的，没有任何 hwirq 和 IRQ 号之间的映射。通过调用 `irq_create_mapping()` 将映射添加到 `irq_domain` 中，该函数接受 `irq_domain` 和一个 hwirq 号作为参数。如果 hwirq 的映射还不存在，那么它将分配一个新的 Linux `irq_desc`，将其与 hwirq 关联起来，并调用 `.map()` 回调，这样驱动程序就可以执行任何必要的硬件设置。

当接收到一个中断时，应该使用 `irq_find_mapping()` 函数从 hwirq 号中找到 Linux IRQ 号。

在调用 `irq_find_mapping()` 之前，至少要调用一次 `irq_create_mapping()` 函数，以免描述符不能被分配。

如果驱动程序有 Linux 的 IRQ 号或 `irq_data` 指针，并且需要知道相关的 hwirq 号（比如在 `irq_chip` 回调中），那么可以直接从 `irq_data->hwirq` 中获得。

### irq\_domain 映射的类型

从 hwirq 到 Linux irq 的反向映射有几种机制，每种机制使用不同的分配函数。应该使用哪种反向映射类型取决于用例。下面介绍每一种反向映射类型：

#### 线性映射

```
irq_domain_add_linear()  
irq_domain_create_linear()
```

线性反向映射维护了一个固定大小的表，该表以 hwirq 号为索引。当一个 hwirq 被映射时，会给 hwirq 分配一个 `irq_desc`，并将 irq 号存储在表中。

当最大的 hwirq 号固定且数量相对较少时，线性图是一个很好的选择 ( $\sim < 256$ )。这种映射的优点是固定时间查找 IRQ 号，而且 `irq_descs` 只分配给在用的 IRQ。缺点是该表必须尽可能大的 hwirq 号。

`irq_domain_add_linear()` 和 `irq_domain_create_linear()` 在功能上是等价的，除了第一个参数不同——前者接受一个 Open Firmware 特定的 ‘`struct device_node`’ 而后者接受一个更通用的抽象 ‘`struct fwnode_handle`’。

大多数驱动应该使用线性映射

## 树状映射

```
irq_domain_add_tree()
irq_domain_create_tree()
```

`irq_domain` 维护着从 hwirq 号到 Linux IRQ 的 radix 的树状映射。当一个 hwirq 被映射时，一个 `irq_desc` 被分配，hwirq 被用作 radix 树的查找键。

如果 hwirq 号可以非常大，树状映射是一个很好的选择，因为它不需要分配一个和最大 hwirq 号一样大的表。缺点是，hwirq 到 IRQ 号的查找取决于表中有多少条目。

`irq_domain_add_tree()` 和 `irq_domain_create_tree()` 在功能上是等价的，除了第一个参数不同——前者接受一个 Open Firmware 特定的 ‘`struct device_node`’，而后者接受一个更通用的抽象 ‘`struct fwnode_handle`’。

很少有驱动应该需要这个映射。

## 无映射

```
irq_domain_add_nomap()
```

当硬件中的 hwirq 号是可编程的时候，就可以采用无映射类型。在这种情况下，最好将 Linux IRQ 号编入硬件本身，这样就不需要映射了。调用 `irq_create_direct_mapping()` 会分配一个 Linux IRQ 号，并调用 `.map()` 回调，这样驱动就可以将 Linux IRQ 号编入硬件中。

大多数驱动程序不能使用这个映射。

## 传统映射类型

```
irq_domain_add_simple()
irq_domain_add_legacy()
irq_domain_add_legacy_isa()
irq_domain_create_simple()
irq_domain_create_legacy()
```

传统映射是已经为 hwirqs 分配了一系列 `irq_descs` 的驱动程序的特殊情况。当驱动程序不能立即转换为使用线性映射时，就会使用它。例如，许多嵌入式系统板卡支持文件使用一组用于 IRQ 号的定义 (`#define`)，这些定义被传递给 `struct device_node` 注册。在这种情况下，不能动态分配 Linux IRQ 号，应该使用传统映射。

传统映射假设已经为控制器分配了一个连续的 IRQ 号范围，并且可以通过向 hwirq 号添加一个固定的偏移来计算 IRQ 号，反之亦然。缺点是需要中断控制器管理 IRQ 分配，并且需要为每个 hwirq 分配一个 irq\_desc，即使它没有被使用。

只有在必须支持固定的 IRQ 映射时，才应使用传统映射。例如，ISA 控制器将使用传统映射来映射 Linux IRQ 0-15，这样现有的 ISA 驱动程序就能得到正确的 IRQ 号。

大多数使用传统映射的用户应该使用 irq\_domain\_add\_simple() 或 irq\_domain\_create\_simple()，只有在系统提供 IRQ 范围时才会使用传统域，否则将使用线性域映射。这个调用的语义是这样的：如果指定了一个 IRQ 范围，那么描述符将被即时分配给它，如果没有范围被分配，它将不会执行 irq\_domain\_add\_linear() 或 irq\_domain\_create\_linear()，这意味着 *no irq* 描述符将被分配。

一个简单域的典型用例是，irqchip 供应商同时支持动态和静态 IRQ 分配。

为了避免最终出现使用线性域而没有描述符被分配的情况，确保使用简单域的驱动程序在任何 irq\_find\_mapping() 之前调用 irq\_create\_mapping() 是非常重要的，因为后者实际上将用于静态 IRQ 分配情况。

irq\_domain\_add\_simple() 和 irq\_domain\_create\_simple() 以及 irq\_domain\_add\_legacy() 和 irq\_domain\_create\_legacy() 在功能上是等价的，只是第一个参数不同-前者接受 Open Firmware 特定的 ‘struct device\_node’，而后者接受一个更通用的抽象 ‘struct fwnode\_handle’。

## IRQ 域层级结构

在某些架构上，可能有多个中断控制器参与将一个中断从设备传送到目标 CPU。让我们来看看 x86 平台上典型的中断传递路径吧

```
Device --> IOAPIC -> Interrupt remapping Controller -> Local APIC -> CPU
```

涉及到的中断控制器有三个：

- 1) IOAPIC 控制器
- 2) 中断重映射控制器
- 3) Local APIC 控制器

为了支持这样的硬件拓扑结构，使软件架构与硬件架构相匹配，为每个中断控制器建立一个 irq\_domain 数据结构，并将这些 irq\_domain 组织成层次结构。

在建立 irq\_domain 层次结构时，靠近设备的 irq\_domain 为子域，靠近 CPU 的 irq\_domain 为父域。所以在上面的例子中，将建立如下的层次结构。

```
CPU Vector irq_domain (root irq_domain to manage CPU vectors)
  ^
  |
Interrupt Remapping irq_domain (manage irq_remapping entries)
  ^
  |
IOAPIC irq_domain (manage IOAPIC delivery entries/pins)
```

使用 irq\_domain 层次结构的主要接口有四个：

- 1) irq\_domain\_alloc\_irqs(): 分配 IRQ 描述符和与中断控制器相关的资源来传递这些中断。
- 2) irq\_domain\_free\_irqs(): 释放 IRQ 描述符和与这些中断相关的中断控制器资源。
- 3) irq\_domain\_activate\_irq(): 激活中断控制器硬件以传递中断。
- 4) irq\_domain\_deactivate\_irq(): 停用中断控制器硬件，停止传递中断。

为了支持 irq\_domain 层次结构，需要做如下修改：

- 1) 一个新的字段 ‘parent’ 被添加到 irq\_domain 结构中；它用于维护 irq\_domain 的层次信息。
- 2) 一个新的字段 ‘parent\_data’ 被添加到 irq\_data 结构中；它用于建立层次结构 irq\_data 以匹配 irq\_domain 层次结构。irq\_data 用于存储 irq\_domain 指针和硬件 irq 号。
- 3) 新的回调被添加到 irq\_domain\_ops 结构中，以支持层次结构的 irq\_domain 操作。

在支持分层 irq\_domain 和分层 irq\_data 准备就绪后，为每个中断控制器建立一个 irq\_domain 结构，并为每个与 IRQ 相关联的 irq\_domain 分配一个 irq\_data 结构。现在我们可以再进一步支持堆栈式（层次结构）的 irq\_chip。也就是说，一个 irq\_chip 与层次结构中的每个 irq\_data 相关联。一个子 irq\_chip 可以自己或通过与它的父 irq\_chip 合作来实现一个所需的操作。

通过堆栈式的 irq\_chip，中断控制器驱动只需要处理自己管理的硬件，在需要的时候可以向其父 irq\_chip 请求服务。所以我们可以实现更简洁的软件架构。

为了让中断控制器驱动程序支持 irq\_domain 层次结构，它需要做到以下几点：

- 1) 实现 irq\_domain\_ops.alloc 和 irq\_domain\_ops.free
- 2) 可选择地实现 irq\_domain\_ops.activate 和 irq\_domain\_ops.deactivate.
- 3) 可选择地实现一个 irq\_chip 来管理中断控制器硬件。
- 4) 不需要实现 irq\_domain\_ops.map 和 irq\_domain\_ops.unmap，它们在层次结构 irq\_domain 中是不用的。

irq\_domain 层次结构绝不是 x86 特有的，大量用于支持其他架构，如 ARM、ARM64 等。

## 调试功能

打开 CONFIG\_GENERIC\_IRQ\_DEBUGFS，可让 IRQ 子系统的大部分内部结构都在 debugfs 中暴露出来。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题, 请联系该文件的译者, 或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/core-api/irq/irqflags-tracing.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

### IRQ-flags 状态追踪

**Author** 最初由 Ingo Molnar <[mingo@redhat.com](mailto:mingo@redhat.com)> 开始撰写

“irq-flags tracing”（中断标志追踪）功能可以“追踪”hardirq 和 softirq 的状态, 它让感兴趣的子系统有机会了解到到内核中发生的每一个 hardirqs-off/hardirqs-on、softirqs-off/softirqs-on 事件。

CONFIG\_TRACE\_IRQFLAGS\_SUPPORT 是通用锁调试代码提供的 CONFIG\_PROVE\_SPIN\_LOCKING 和 CONFIG\_PROVE\_RW\_LOCKING 所需要的。否则将只有 CONFIG\_PROVE\_MUTEX\_LOCKING 和 CONFIG\_PROVE\_RWSEM\_LOCKING 在一个架构上被提供-这些都是不在 IRQ 上下文中使用的锁 API。（rwsems 的一个异常是可以解决的）

架构对这一点的支持当然不属于“微不足道”的范畴, 因为很多低级的汇编代码都要处理 irq-flags 的状态变化。但是一个架构可以以一种相当直接且无风险的方式启用 irq-flags-tracing。

架构如果想支持这个, 需要先做一些代码组织上的改变:

- 在他们的 arch 级 Kconfig 文件中添加并启用 TRACE\_IRQFLAGS\_SUPPORT。

然后还需要做一些功能上的改变来实现对 irq-flags-tracing 的支持:

- 在低级入口代码中增加（构建条件）对 trace\_hardirqs\_off()/trace\_hardirqs\_on() 函数的调用。锁验证器会密切关注“real”的irq-flags 是否与“virtual”的irq-flags 状态相匹配, 如果两者不匹配, 则会发出警告（并关闭自己）。通常维护 arch 中 irq-flags-track 的大部分时间都是在这种状态下度过的: 看看 lockdep 的警告, 试着找出我们还没有搞定的汇编代码。修复并重复。一旦系统启动, 并且在 irq-flags 跟踪功能中没有出现 lockdep 警告的情况下, arch 支持就完成了。
- 如果该架构有不可屏蔽的中断, 那么需要通过 lockdep\_off()/lockdep\_on() 将这些中断从 irq 跟踪 [和锁验证] 机制中排除。

一般来说, 在一个架构中, 不完整的 irq-flags-tracing 实现是没有风险的: lockdep 会检测到这一点, 并将自己关闭。即锁验证器仍然可靠。应该不会因为 irq-tracing 的错误而崩溃。(除非通过修改不该修改的条件来更改汇编或寄存器而破坏其他代码)

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解, 而不是作为一个分支。因此, 如果您对此文件有任何意见或更新, 请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题, 请联系该文件的译者, 或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/core-api/refcount-vs-atomic.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## 与 atomic\_t 相比, refcount\_t 的 API 是这样的

- 简介
- 内存顺序的相关类型
- 函数的比较
  - 情况 1) - 非“读/修改/写”(RMW) 操作
  - 情况 2) - 基于增量的操作, 不返回任何值
  - 情况 3) - 基于递减的 RMW 操作, 没有返回值
  - 情况 4) - 基于增量的 RMW 操作, 返回一个值
  - 情况 5) - 基于 Dec/Sub 递减的通用 RMW 操作, 返回一个值
  - 情况 6) 其他基于递减的 RMW 操作, 返回一个值
  - 情况 7) - 基于锁的 RMW

## 简介

refcount\_t API 的目标是为实现对象的引用计数器提供一个最小的 API。虽然来自 lib/refcount.c 的独立于架构的通用实现在下面使用了原子操作, 但一些 `refcount_*`() 和 `atomic_*`() 函数在内存顺序保证方面有很多不同。本文档概述了这些差异, 并提供了相应的例子, 以帮助开发者根据这些内存顺序保证的变化来验证他们的代码。

本文档中使用的术语尽量遵循 tools/memory-model/Documentation/explanation.txt 中定义的正式 LKMM。

`memory-barriers.txt` 和 `atomic_t.txt` 提供了更多关于内存顺序的背景, 包括通用的和针对原子操作的。

## 内存顺序的相关类型

**Note:** 下面的部分只涵盖了本文使用的与原子操作和引用计数器有关的一些内存顺序类型。如果想了解更广泛的情况，请查阅 `memory-barriers.txt` 文件。

在没有任何内存顺序保证的情况下（即完全无序），`atomics` 和 `refcounters` 只提供原子性和程序顺序（program order, po）关系（在同一个 CPU 上）。它保证每个 `atomic_*` () 和 `refcount_*`() 操作都是原子性的，指令在单个 CPU 上按程序顺序执行。这是用 `READ_ONCE()/WRITE_ONCE()` 和比较并交换原语实现的。

强（完全）内存顺序保证在同一 CPU 上的所有较早加载和存储的指令（所有程序顺序较早 [po-earlier] 指令）在执行任何程序顺序较后指令（po-later）之前完成。它还保证同一 CPU 上储存的程序优先较早的指令和来自其他 CPU 传播的指令必须在该 CPU 执行任何程序顺序较后指令之前传播到其他 CPU (A-累积属性)。这是用 `smp_mb()` 实现的。

`RELEASE` 内存顺序保证了在同一 CPU 上所有较早加载和存储的指令（所有程序顺序较早指令）在此操作前完成。它还保证同一 CPU 上储存的程序优先较早的指令和来自其他 CPU 传播的指令必须在释放（release）操作之前传播到所有其他 CPU (A-累积属性)。这是用 `smp_store_release()` 实现的。

`ACQUIRE` 内存顺序保证了同一 CPU 上的所有后加载和存储的指令（所有程序顺序较后指令）在获取（acquire）操作之后完成。它还保证在获取操作执行后，同一 CPU 上储存的所有程序顺序较后指令必须传播到所有其他 CPU。这是用 `smp_acquire_after_ctrl_dep()` 实现的。

对 `Refcounters` 的控制依赖（取决于成功）保证了如果一个对象的引用被成功获得（引用计数器的增量或增加行为发生了，函数返回 `true`），那么进一步的存储是针对这个操作的命令。对存储的控制依赖没有使用任何明确的屏障来实现，而是依赖于 CPU 不对存储进行猜测。这只是一个单一的 CPU 关系，对其他 CPU 不提供任何保证。

## 函数的比较

### 情况 1) - 非“读/修改/写”(RMW) 操作

函数变化：

- `atomic_set() -> refcount_set()`
- `atomic_read() -> refcount_read()`

内存顺序保证变化：

- `none` (两者都是完全无序的)

## 情况 2) - 基于增量的操作, 不返回任何值

函数变化:

- atomic\_inc() -> refcount\_inc()
- atomic\_add() -> refcount\_add()

内存顺序保证变化:

- none (两者都是完全无序的)

## 情况 3) - 基于递减的 RMW 操作, 没有返回值

函数变化:

- atomic\_dec() -> refcount\_dec()

内存顺序保证变化:

- 完全无序的 -> RELEASE 顺序

## 情况 4) - 基于增量的 RMW 操作, 返回一个值

函数变化:

- atomic\_inc\_not\_zero() -> refcount\_inc\_not\_zero()
- 无原子性对应函数 -> refcount\_add\_not\_zero()

内存顺序保证变化:

- 完全有序的 -> 控制依赖于存储的成功

**Note:** 此处 假设了, 必要的顺序是作为获得对象指针的结果而提供的。

## 情况 5) - 基于 Dec/Sub 递减的通用 RMW 操作, 返回一个值

函数变化:

- atomic\_dec\_and\_test() -> refcount\_dec\_and\_test()
- atomic\_sub\_and\_test() -> refcount\_sub\_and\_test()

内存顺序保证变化:

- 完全有序的 -> RELEASE 顺序 + 成功后 ACQUIRE 顺序

## 情况 6) 其他基于递减的 RMW 操作，返回一个值

函数变化:

- 无原子性对应函数 -> `refcount_dec_if_one()`
- `atomic_add_unless(&var, -1, 1)` -> `refcount_dec_not_one(&var)`

内存顺序保证变化:

- 完全有序的 -> RELEASE 顺序 + 控制依赖

---

**Note:** `atomic_add_unless()` 只在执行成功时提供完整的顺序。

---

## 情况 7) -基于锁的 RMW

函数变化:

- `atomic_dec_and_lock()` -> `refcount_dec_and_lock()`
- `atomic_dec_and_mutex_lock()` -> `refcount_dec_and_mutex_lock()`

内存顺序保证变化:

- 完全有序 -> RELEASE 顺序 + 控制依赖 + 持有

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/core-api/local\_ops.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## 本地原子操作的语义和行为

作者 Mathieu Desnoyers

本文解释了本地原子操作的目的，如何为任何给定的架构实现这些操作，并说明了如何正确使用这些操作。它还强调了在内存写入顺序很重要的情况下，跨 CPU 读取这些本地变量时必须采取的预防措施。

**Note:** 注意，基于 `local_t` 的操作不建议用于一般内核操作。请使用 `this_cpu` 操作来代替使用，除非真的有特殊目的。大多数内核中使用的 `local_t` 已经被 `this_cpu` 操作所取代。`this_cpu` 操作在一条指令中结合了重定位和类似 `local_t` 的语义，产生了更紧凑和更快的执行代码。

## 本地原子操作的目的

本地原子操作的目的是提供快速和高度可重入的每 CPU 计数器。它们通过移除 LOCK 前缀和通常需要在 CPU 间同步的内存屏障，将标准原子操作的性能成本降到最低。

在许多情况下，拥有快速的每 CPU 原子计数器是很有吸引力的：它不需要禁用中断来保护中断处理程序，它允许在 NMI (Non Maskable Interrupt) 处理程序中使用连贯的计数器。它对追踪目的和各种性能监测计数器特别有用。

本地原子操作只保证在拥有数据的 CPU 上的变量修改的原子性。因此，必须注意确保只有一个 CPU 写到 `local_t` 的数据。这是通过使用每 CPU 的数据来实现的，并确保我们在一个抢占式安全上下文中修改它。然而，从任何一个 CPU 读取 `local_t` 数据都是允许的：这样它就会显得与所有者 CPU 的其他内存写入顺序不一致。

## 针对特定架构的实现

这可以通过稍微修改标准的原子操作来实现：只有它们的 UP 变体必须被保留。这通常意味着删除 LOCK 前缀（在 i386 和 x86\_64 上）和任何 SMP 同步屏障。如果架构在 SMP 和 UP 之间没有不同的行为，在你的架构的 `local.h` 中包括 `asm-generic/local.h` 就足够了。

通过在一个结构体中嵌入一个 `atomic_long_t`，`local_t` 类型被定义为一个不透明的 `signed long`。这样做的目的是为了使从这个类型到 `long` 的转换失败。该定义看起来像：

```
typedef struct { atomic_long_t a; } local_t;
```

## 使用本地原子操作时应遵循的规则

- 被本地操作触及的变量必须是每 cpu 的变量。
- 只有这些变量的 CPU 所有者才可以写入这些变量。
- 这个 CPU 可以从任何上下文（进程、中断、软中断、nmi…）中使用本地操作来更新它的 local\_t 变量。
- 当在进程上下文中使用本地操作时，必须禁用抢占（或中断），以确保进程在获得每 CPU 变量和进行实际的本地操作之间不会被迁移到不同的 CPU。
- 当在中断上下文中使用本地操作时，在主线内核上不需要特别注意，因为它们将在局部 CPU 上运行，并且已经禁用了抢占。然而，我建议无论如何都要明确地禁用抢占，以确保它在-rt 内核上仍能正确工作。
- 读取本地 cpu 变量将提供该变量的当前拷贝。
- 对这些变量的读取可以从任何 CPU 进行，因为对“long”，对齐的变量的更新总是原子的。由于写入程序的 CPU 没有进行内存同步，所以在读取 其他 cpu 的变量时，可以读取该变量的过期副本。

## 如何使用本地原子操作

```
#include <linux/percpu.h>
#include <asm/local.h>

static DEFINE_PER_CPU(local_t, counters) = LOCAL_INIT(0);
```

### 计数器

计数是在一个 signed long 的所有位上进行的。

在可抢占的上下文中，围绕本地原子操作使用 `get_cpu_var()` 和 `put_cpu_var()`：它确保在对每个 cpu 变量进行写访问时，抢占被禁用。比如说：

```
local_inc(&get_cpu_var(counters));
put_cpu_var(counters);
```

如果你已经在抢占安全上下文中，你可以使用 `this_cpu_ptr()` 代替：

```
local_inc(this_cpu_ptr(&counters));
```

## 读取计数器

那些本地计数器可以从外部的 CPU 中读取，以求得计数的总和。请注意，`local_read` 所看到的跨 CPU 的数据必须被认为是相对于拥有该数据的 CPU 上发生的其他内存写入来说不符合顺序的：

```
long sum = 0;
for_each_online_cpu(cpu)
    sum += local_read(&per_cpu(counters, cpu));
```

如果你想使用远程 `local_read` 来同步 CPU 之间对资源的访问，必须在写入者和读取者的 CPU 上分别使用显式的 `smp_wmb()` 和 `smp_rmb()` 内存屏障。如果你使用 `local_t` 变量作为写在缓冲区中的字节的计数器，就会出现这种情况：在缓冲区写和计数器增量之间应该有一个 `smp_wmb()`，在计数器读和缓冲区读之间也应有一个 `smp_rmb()`。

下面是一个使用 `local.h` 实现每个 cpu 基本计数器的示例模块：

```
/* test-local.c
 *
 * Sample module for local.h usage.
 */

#include <asm/local.h>
#include <linux/module.h>
#include <linux/timer.h>

static DEFINE_PER_CPU(local_t, counters) = LOCAL_INIT(0);

static struct timer_list test_timer;

/* IPI called on each CPU. */
static void test_each(void *info)
{
    /* Increment the counter from a non preemptible context */
    printk("Increment on cpu %d\n", smp_processor_id());
    local_inc(this_cpu_ptr(&counters));

    /* This is what incrementing the variable would look like within a
     * preemptible context (it disables preemption) :
     *
     * local_inc(&get_cpu_var(counters));
     * put_cpu_var(counters);
     */
}

static void do_test_timer(unsigned long data)
{
    int cpu;

    /* Increment the counters */
```

```
on_each_cpu(test_each, NULL, 1);
/* Read all the counters */
printk("Counters read from CPU %d\n", smp_processor_id());
for_each_online_cpu(cpu) {
    printk("Read : CPU %d, count %ld\n", cpu,
           local_read(&per_cpu(counters, cpu)));
}
mod_timer(&test_timer, jiffies + 1000);
}

static int __init test_init(void)
{
    /* initialize the timer that will increment the counter */
    timer_setup(&test_timer, do_test_timer, 0);
    mod_timer(&test_timer, jiffies + 1);

    return 0;
}

static void __exit test_exit(void)
{
    del_timer_sync(&test_timer);
}

module_init(test_init);
module_exit(test_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Mathieu Desnoyers");
MODULE_DESCRIPTION("Local Atomic Ops");
```

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/core-api/padata.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## padata 并行执行机制

日期 2020 年 5 月

Padata 是一种机制，内核可以通过此机制将工作分散到多个 CPU 上并行完成，同时可以选择保持它们的顺序。

它最初是为 IPsec 开发的，它需要在不对这些数据包重新排序的前提下，为大量的数据包进行加密和解密。这是目前 padata 的序列化作业支持的唯一用途。

Padata 还支持多线程作业，将作业平均分割，同时在线程之间进行负载均衡和协调。

## 执行序列化作业

### 初始化

使用 padata 执行序列化作业的第一步是建立一个 padata\_instance 结构体，以全面控制作业的运行方式：

```
#include <linux/padata.h>

struct padata_instance *padata_alloc(const char *name);
```

‘name’ 即标识了这个实例。

然后，通过分配一个 padata\_shell 来完成 padata 的初始化：

```
struct padata_shell *padata_alloc_shell(struct padata_instance *pinst);
```

一个 padata\_shell 用于向 padata 提交一个作业，并允许一系列这样的作业被独立地序列化。一个 padata\_instance 可以有一个或多个 padata\_shell 与之相关联，每个都允许一系列独立的作业。

### 修改 cpumasks

用于运行作业的 CPU 可以通过两种方式改变，通过 padata\_set\_cpumask() 编程或通过 sysfs。前者的定义是：

```
int padata_set_cpumask(struct padata_instance *pinst, int cpumask_type,
                      cpumask_var_t cpumask);
```

这里 cpumask\_type 是 PADATA\_CPU\_PARALLEL (并行) 或 PADATA\_CPU\_SERIAL (串行) 之一，其中并行 cpumask 描述了哪些处理器将被用来并行执行提交给这个实例的作业，串行 cpumask 定义了哪些处理器被允许用作串行化回调处理器。cpumask 指定了要使用的新 cpumask。

一个实例的 cpumasks 可能有 sysfs 文件。例如，pcrypt 的文件在 /sys/kernel/pcrypt/<instance-name>。在一个实例的目录中，有两个文件，parallel\_cpumask 和 serial\_cpumask，任何一个 cpumask 都可以通过在文件中回显 (echo) 一个 bitmask 来改变，比如说：

```
echo f > /sys/kernel/pcrypt/pencrypt/parallel_cpumask
```

读取其中一个文件会显示用户提供的 cpumask，它可能与“可用”的 cpumask 不同。

Padata 内部维护着两对 cpumask，用户提供的 cpumask 和“可用的” cpumask(每一对由一个并行和一个串行 cpumask 组成)。用户提供的 cpumasks 在实例分配时默认为所有可能的 CPU，并且可以如上所述进行更改。可用的 cpumasks 总是用户提供的 cpumasks 的一个子集，只包含用户提供的掩码中的在线 CPU；这些是 pdata 实际使用的 cpumasks。因此，向 pdata 提供一个包含离线 CPU 的 cpumask 是合法的。一旦用户提供的 cpumask 中的一个离线 CPU 上线，pdata 就会使用它。

改变 CPU 掩码的操作代价很高，所以不应频繁更改。

## 运行一个作业

实际上向 pdata 实例提交工作需要创建一个 pdata\_priv 结构体，它代表一个作业：

```
struct pdata_priv {
    /* Other stuff here... */
    void (*parallel)(struct pdata_priv *pdata);
    void (*serial)(struct pdata_priv *pdata);
};
```

这个结构体几乎肯定会被嵌入到一些针对要做的工作的大结构体中。它的大部分字段对 pdata 来说是私有的，但是这个结构在初始化时应该被清零，并且应该提供 parallel() 和 serial() 函数。在完成工作的过程中，这些函数将被调用，我们马上就会遇到。

工作的提交是通过：

```
int pdata_do_parallel(struct pdata_shell *ps,
                      struct pdata_priv *pdata, int *cb_cpu);
```

ps 和 pdata 结构体必须如上所述进行设置；cb\_cpu 指向作业完成后用于最终回调的首选 CPU；它必须在当前实例的 CPU 掩码中（如果不是，cb\_cpu 指针将被更新为指向实际选择的 CPU）。pdata\_do\_parallel() 的返回值在成功时为 0，表示工作正在进行中。-EBUSY 意味着有人在其他地方正在搞乱实例的 CPU 掩码，而当 cb\_cpu 不在串行 cpumask 中、并行或串行 cpumasks 中无在线 CPU，或实例停止时，则会出现-EINVAL 反馈。

每个提交给 pdata\_do\_parallel() 的作业将依次传递给一个 CPU 上的上述 parallel() 函数的一个调用，所以真正的并行是通过提交多个作业来实现的。parallel() 在运行时禁用软件中断，因此不能睡眠。parallel() 函数把获得的 pdata\_priv 结构体指针作为其唯一的参数；关于实际要做的工作的信息可能是通过使用 container\_of() 找到封装结构体来获得的。

请注意，parallel() 没有返回值；pdata 子系统假定 parallel() 将从此时开始负责这项工作。作业不需要在这次调用中完成，但是，如果 parallel() 留下了未完成的工作，它应该准备在前一个作业完成之前，被以新的作业再次调用。

## 序列化作业

当一个作业完成时，parallel()（或任何实际完成该工作的函数）应该通过调用通知 pdata 此事：

```
void pdata_do_serial(struct pdata_priv *pdata);
```

在未来的某个时刻，pdata\_do\_serial() 将触发对 pdata\_priv 结构体中 serial() 函数的调用。这个调用将发生在最初要求调用 pdata\_do\_parallel() 的 CPU 上；它也是在本地软件中断被禁用的情况下运行的。请注意，这个调用可能会被推迟一段时间，因为 pdata 代码会努力确保作业按照提交的顺序完成。

## 销毁

清理一个 pdata 实例时，可以预见的是调用两个 free 函数，这两个函数对应于分配的逆过程：

```
void pdata_free_shell(struct pdata_shell *ps);
void pdata_free(struct pdata_instance *pinst);
```

用户有责任确保在调用上述任何一项之前，所有未完成的工作都已完成。

## 运行多线程作业

一个多线程作业有一个主线程和零个或多个辅助线程，主线程参与作业，然后等待所有辅助线程完成。pdata 将作业分割成称为 chunk 的单元，其中 chunk 是一个线程在一次调用线程函数中完成的作业片段。

用户必须做三件事来运行一个多线程作业。首先，通过定义一个 pdata\_mt\_job 结构体来描述作业，这在接口部分有解释。这包括一个指向线程函数的指针，pdata 每次将作业块分配给线程时都会调用这个函数。然后，定义线程函数，它接受三个参数：start、end 和 arg，其中前两个参数限定了线程操作的范围，最后一个是指向作业共享状态的指针，如果有的话。准备好共享状态，它通常被分配在主线程的堆栈中。最后，调用 pdata\_do\_multithreaded()，它将在作业完成后返回。

## 接口

该 API 在以下内核代码中：

include/linux/pdata.h

kernel/pdata.c

Todolist:

..../RCU/index

### 低级硬件管理

缓存管理, CPU 热插拔管理等。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解, 而不是作为一个分支。因此, 如果您对此文件有任何意见或更新, 请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题, 请联系该文件的译者, 或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/core-api/cachetlb.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

### Linux 下的缓存和 TLB 刷新

作者 David S. Miller <[davem@redhat.com](mailto:davem@redhat.com)>

译注: TLB, Translation Lookaside Buffer, 页表缓存/变换旁查缓冲器

本文描述了由 Linux 虚拟内存子系统调用的缓存/TLB 刷新接口。它列举了每个接口, 描述了它的预期目的, 以及接口被调用后的预期副作用。

下面描述的副作用是针对单处理器的实现, 以及在单个处理器上发生的情况。若为 SMP, 则只需将定义简单地扩展一下, 使发生在某个特定接口的副作用扩展到系统的所有处理器上。不要被这句话吓到, 以为 SMP 的缓存/tlb 刷新一定是很低效的, 事实上, 这是一个可以进行很多优化的领域。例如, 如果可以证明一个用户地址空间从未在某个 cpu 上执行过 (见 mm\_cpumask()), 那么就不需要在该 cpu 上对这个地址空间进行刷新。

首先是 TLB 刷新接口, 因为它们是最简单的。在 Linux 下, TLB 被抽象为 cpu 用来缓存从软件页表获得的虚拟-> 物理地址转换的东西。这意味着, 如果软件页表发生变化, 这个“TLB”缓存中就有可能出现过时(脏)的翻译。因此, 当软件页表发生变化时, 内核会在页表发生变化后调用以下一种刷新方法:

1) `void flush_tlb_all(void)`

最严格的刷新。在这个接口运行后, 任何以前的页表修改都会对 cpu 可见。

这通常是在内核页表被改变时调用的, 因为这种转换在本质上是“全局”的。

2) `void flush_tlb_mm(struct mm_struct *mm)`

这个接口从 TLB 中刷新整个用户地址空间。在运行后, 这个接口必须确保以前对地址空间‘mm’的任何页表修改对 cpu 来说是可见的。也就是说, 在运行后, TLB 中不会有‘mm’的页表项。

这个接口被用来处理整个地址空间的页表操作, 比如在 fork 和 exec 过程中发生的事情。

## 3) void flush\_tlb\_range(struct vm\_area\_struct \*vma, unsigned long start, unsigned long end)

这里我们要从 TLB 中刷新一个特定范围的（用户）虚拟地址转换。在运行后，这个接口必须确保以前对 ‘start’ 到 ‘end-1’ 范围内的地址空间 ‘vma->vm\_mm’ 的任何页表修改对 cpu 来说是可见的。也就是说，在运行后，TLB 中不会有 ‘mm’ 的页表项用于 ‘start’ 到 ‘end-1’ 范围内的虚拟地址。

“vma” 是用于该区域的备份存储。主要是用于 munmap() 类型的操作。

提供这个接口是希望端口能够找到一个合适的有效方法来从 TLB 中删除多个页面大小的转换，而不是让内核为每个可能被修改的页表项调用 flush\_tlb\_page(见下文)。

## 4) void flush\_tlb\_page(struct vm\_area\_struct \*vma, unsigned long addr)

这一次我们需要从 TLB 中删除 PAGE\_SIZE 大小的转换。‘vma’ 是 Linux 用来跟踪进程的 mmap 区域的支持结构体，地址空间可以通过 vma->vm\_mm 获得。另外，可以通过测试 (vma->vm\_flags & VM\_EXEC) 来查看这个区域是否是可执行的（因此在 split-tlb 类型的设置中可能在“指令 TLB”中）。

在运行后，这个接口必须确保之前对用户虚拟地址 “addr”的地址空间 “vma->vm\_mm”的页表修改对 cpu 来说是可见的。也就是说，在运行后，TLB 中不会有虚拟地址 ‘addr’ 的 ‘vma->vm\_mm’ 的页表项。

这主要是在故障处理时使用。

## 5) void update\_mmu\_cache(struct vm\_area\_struct \*vma, unsigned long address, pte\_t \*ptep)

在每个缺页异常结束时，这个程序被调用，以告诉体系结构特定的代码，在软件页表中，在地址空间 “vma->vm\_mm”的虚拟地址 “地址” 处，现在存在一个翻译。

可以用它所选择的任何方式使用这个信息来进行移植。例如，它可以使用这个事件来为软件管理的 TLB 配置预装 TLB 转换。目前 sparc64 移植就是这么干的。

接下来，我们有缓存刷新接口。一般来说，当 Linux 将现有的虚拟-> 物理映射改变为新的值时，其顺序将是以下形式之一：

- 1) flush\_cache\_mm(mm);  
change\_all\_page\_tables\_of(mm);  
flush\_tlb\_mm(mm);
- 2) flush\_cache\_range(vma, start, end);  
change\_range\_of\_page\_tables(mm, start, end);  
flush\_tlb\_range(vma, start, end);
- 3) flush\_cache\_page(vma, addr, pfn);  
set\_pte(pte\_pointer, new\_pte\_val);  
flush\_tlb\_page(vma, addr);

缓存级别的刷新将永远是第一位的，因为这允许我们正确处理那些缓存严格，且在虚拟地址被从缓存中刷新时要求一个虚拟地址的虚拟-> 物理转换存在的系统。HyperSparc cpu 就是这样一个具有这种属性的 cpu。

下面的缓存刷新程序只需要在特定的 cpu 需要的范围内处理缓存刷新。大多数情况下，这些程序必须为 cpu 实现，这些 cpu 有虚拟索引的缓存，当虚拟-> 物理转换被改变或移除时，必须被刷新。因此，例如，IA32 处理器的物理索引的物理标记的缓存没有必要实现这些接口，因为这些缓存是完全同步的，并且不依赖于翻译信息。

下面逐个列出这些程序：

### 1) void flush\_cache\_mm(struct mm\_struct \*mm)

这个接口将整个用户地址空间从高速缓存中刷掉。也就是说，在运行后，将没有与 ‘mm’ 相关的缓存行。

这个接口被用来处理整个地址空间的页表操作，比如在退出和执行过程中发生的事情。

### 2) void flush\_cache\_dup\_mm(struct mm\_struct \*mm)

这个接口将整个用户地址空间从高速缓存中刷新掉。也就是说，在运行后，将没有与 ‘mm’ 相关的缓存行。

这个接口被用来处理整个地址空间的页表操作，比如在 fork 过程中发生的事情。

这个选项与 flush\_cache\_mm 分开，以允许对 VIPT 缓存进行一些优化。

### 3) void flush\_cache\_range(struct vm\_area\_struct \*vma, unsigned long start, unsigned long end)

在这里，我们要从缓存中刷新一个特定范围的（用户）虚拟地址。运行后，在 “start” 到 “end-1” 范围内的虚拟地址的 “vma->vm\_mm”的缓存中将没有页表项。

“vma” 是被用于该区域的备份存储。主要是用于 munmap() 类型的操作。

提供这个接口是希望端口能够找到一个合适的有效方法来从缓存中删除多个页面大小的区域，而不是让内核为每个可能被修改的页表项调用 flush\_cache\_page (见下文)。

### 4) void flush\_cache\_page(struct vm\_area\_struct \*vma, unsigned long addr, unsigned long pfn)

这一次我们需要从缓存中删除一个 PAGE\_SIZE 大小的区域。“vma” 是 Linux 用来跟踪进程的 mmap 区域的支持结构体，地址空间可以通过 vma->vm\_mm 获得。另外，我们可以通过测试 (vma->vm\_flags & VM\_EXEC) 来查看这个区域是否是可执行的（因此在 “Harvard” 类型的缓存布局中可能是在 “指令缓存” 中）。

“pfn” 表示 “addr” 所对应的物理页框（通过 PAGE\_SHIFT 左移这个值来获得物理地址）。正是这个映射应该从缓存中删除。

在运行之后，对于虚拟地址 ‘addr’ 的 ‘vma->vm\_mm’，在缓存中不会有任何页表项，它被翻译成 ‘pfn’。

这主要是在故障处理过程中使用。

### 5) void flush\_cache\_kmaps(void)

只有在平台使用高位内存的情况下才需要实现这个程序。它将在所有的 kmaps 失效之前被调用。

运行后，内核虚拟地址范围 PKMAP\_ADDR(0) 到 PKMAP\_ADDR(LAST\_PKMAP) 的缓存中将没有页表项。

这个程序应该在 `asm/highmem.h` 中实现。

```
6) void flush_cache_vmap(unsigned long start, unsigned long end)           void
   flush_cache_vunmap(unsigned long start, unsigned long end)
```

在这里，在这两个接口中，我们从缓存中刷新一个特定范围的（内核）虚拟地址。运行后，在“start”到“end-1”范围内的虚拟地址的内核地址空间的缓存中不会有页表项。

这两个程序中的第一个是在 `vmap_range()` 安装了页表项之后调用的。第二个是在 `vunmap_range()` 删除页表项之前调用的。

还有一类 cpu 缓存问题，目前需要一套完全不同的接口来正确处理。最大的问题是处理器的数据缓存中的虚拟别名。

**Note:** 这段内容有些晦涩，为了减轻中文阅读压力，特作此译注。

别名（alias）属于缓存一致性问题，当不同的虚拟地址映射相同的物理地址，而这些虚拟地址的 index 不同，此时就发生了别名现象（多个虚拟地址被称为别名）。通俗点来说就是指同一个物理地址的数据被加载到不同的 cacheline 中就会出现别名现象。

常见的解决方法有两种：第一种是硬件维护一致性，设计特定的 cpu 电路来解决问题（例如设计为 PIPT 的 cache）；第二种是软件维护一致性，就是下面介绍的 sparc 的解决方案——页面染色，涉及的技术细节太多，译者不便展开，请读者自行查阅相关资料。

您的移植是否容易在其 D-cache 中出现虚拟别名？嗯，如果您的 D-cache 是虚拟索引的，且 cache 大于 `PAGE_SIZE`（页大小），并且不能防止同一物理地址的多个 cache 行同时存在，您就会遇到这个问题。

如果你的 D-cache 有这个问题，首先正确定义 `asm/shmparam.h` SHMLBA，它基本上应该是你的虚拟寻址 D-cache 的大小（或者如果大小是可变的，则是最大的可能大小）。这个设置将迫使 SYSv IPC 层只允许用户进程在这个值的倍数的地址上对共享内存进行映射。

**Note:** 这并不能解决共享 mmaps 的问题，请查看 sparc64 移植解决这个问题的一个方法（特别是 `SPARC_FLAG_MMAPSHARED`）。

接下来，你必须解决所有其他情况下的 D-cache 别名问题。请记住这个事实，对于一个给定的页面映射到某个用户地址空间，总是至少还有一个映射，那就是内核在其线性映射中从 `PAGE_OFFSET` 开始。因此，一旦第一个用户将一个给定的物理页映射到它的地址空间，就意味着 D-cache 的别名问题有可能存在，因为内核已经将这个页映射到它的虚拟地址。

```
void copy_user_page(void *to, void *from, unsigned long addr, struct page *page) void
clear_user_page(void *to, unsigned long addr, struct page *page)
```

这两个程序在用户匿名或 COW 页中存储数据。它允许一个端口有效地避免用户空间和内核之间的 D-cache 别名问题。

例如，一个端口可以在复制过程中把“from”和“to”暂时映射到内核的虚拟地址上。这两个页面的虚拟地址的选择方式是，内核的加载/存储指令发生在虚拟地址上，而这些虚拟地址与用户的页面映射是相同的“颜色”。例如，Sparc64 就使用这种技术。

“addr”参数告诉了用户最终要映射这个页面的虚拟地址，“page”参数给出了一个指向目标页结构体的指针。

如果 D-cache 别名不是问题，这两个程序可以简单地直接调用 memcpy/memset 而不做其他事情。

```
void flush_dcache_page(struct page *page)
```

任何时候，当内核写到一个页面缓存页，或者内核要从一个页面缓存页中读出，并且这个页面的用户空间共享/可写映射可能存在时，这个程序就会被调用。

---

**Note:** 这个程序只需要为有可能被映射到用户进程的地址空间的页面缓存调用。因此，例如，处理页面缓存中 vfs 符号链接的 VFS 层代码根本不需要调用这个接口。

---

“内核写入页面缓存的页面”这句话的意思是，具体来说，内核执行存储指令，在该页面的 page-> 虚拟映射处弄脏该页面的数据。在这里，通过刷新的手段处理 D-cache 的别名是很重要的，以确保这些内核存储对该页的用户空间映射是可见的。

推论的情况也同样重要，如果有用户对这个文件有共享 + 可写的映射，我们必须确保内核对这些页面的读取会看到用户所做的最新的存储。

如果 D-cache 别名不是一个问题，这个程序可以简单地定义为该架构上的 nop。

在 page->flags (PG\_arch\_1) 中有一个位是“架构私有”。内核保证，对于分页缓存的页面，当这样的页面第一次进入分页缓存时，它将清除这个位。

这使得这些接口可以更有效地被实现。如果目前没有用户进程映射这个页面，它允许我们“推迟”（也许是无限期）实际的刷新过程。请看 sparc64 的 flush\_dcache\_page 和 update\_mmu\_cache 实现，以了解如何做到这一点。

这个想法是，首先在 flush\_dcache\_page() 时，如果 page->mapping->i\_mmap 是一个空树，只需标记架构私有页标志位。之后，在 update\_mmu\_cache() 中，会对这个标志位进行检查，如果设置了，就进行刷新，并清除标志位。

---

**Important:** 通常很重要的是，如果你推迟刷新，实际的刷新发生在同一个 CPU 上，因为它将 cpu 存储到页面上，使其变脏。同样，请看 sparc64 关于如何处理这个问题的例子。

---

```
void copy_to_user_page(struct vm_area_struct *vma, struct page *page, unsigned
long user_vaddr, void *dst, void *src, int len)           void copy_from_user_page(struct
vm_area_struct *vma, struct page *page, unsigned long user_vaddr, void *dst, void
*src, int len)
```

当内核需要复制任意的数据进出任意的用户页时（比如 `ptrace()`），它将使用这两个程序。

任何必要的缓存刷新或其他需要发生的一致性操作都应该在这里发生。如果处理器的指令缓存没有对 cpu 存储进行窥探，那么你很可能需要为 `copy_to_user_page()` 刷新指令缓存。

```
void flush_anon_page(struct vm_area_struct *vma, struct page *page, unsigned long
vmaddr)
```

当内核需要访问一个匿名页的内容时，它会调用这个函数（目前只有 `get_user_pages()`）。注意：`flush_dcache_page()` 故意对匿名页不起作用。默认的实现是 `nop`（对于所有相干的架构应该保持这样）。对于不一致性的架构，它应该刷新 `vmaddr` 处的页面缓存。

```
void flush_icache_range(unsigned long start, unsigned long end)
```

当内核存储到它将执行的地址中时（例如在加载模块时），这个函数被调用。

如果 `icache` 不对存储进行窥探，那么这个程序将需要对其进行刷新。

```
void flush_icache_page(struct vm_area_struct *vma, struct page *page)
```

`flush_icache_page` 的所有功能都可以在 `flush_dcache_page` 和 `update_mmu_cache` 中实现。在未来，我们希望能够完全删除这个接口。

最后一类 API 是用于 I/O 到内核内特意设置的别名地址范围。这种别名是通过使用 `vmap/vmalloc` API 设置的。由于内核 I/O 是通过物理页进行的，I/O 子系统假定用户映射和内核偏移映射是唯一的别名。这对 `vmap` 别名来说是不正确的，所以内核中任何试图对 `vmap` 区域进行 I/O 的东西都必须手动管理一致性。它必须在做 I/O 之前刷新 `vmap` 范围，并在 I/O 返回后使其失效。

```
void flush_kernel_vmap_range(void *vaddr, int size)
```

刷新 `vmap` 区域中指定的虚拟地址范围的内核缓存。这是为了确保内核在 `vmap` 范围内修改的任何数据对物理页是可见的。这个设计是为了使这个区域可以安全地执行 I/O。注意，这个 API 并没有刷新该区域的偏移映射别名。

```
void invalidate_kernel_vmap_range(void *vaddr, int size) invalidates
```

在 `vmap` 区域的一个给定的虚拟地址范围的缓存，这可以防止处理器在物理页的 I/O 发生时通过投机性地读取数据而使缓存变脏。这只对读入 `vmap` 区域的数据是必要的。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/core-api/cpu\_hotplug.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 吴想成 Wu XiangCheng <[bowxc@email.cn](mailto:bowxc@email.cn)>

### 内核中的 CPU 热拔插

时间 2016 年 12 月

作者 Sebastian Andrzej Siewior <[bigeasy@linutronix.de](mailto:bigeasy@linutronix.de)>, Rusty Russell <[rusty@rustcorp.com.au](mailto:rusty@rustcorp.com.au)>, Srivatsa Vaddagiri <[vatsa@in.ibm.com](mailto:vatsa@in.ibm.com)>, Ashok Raj <[ashok.raj@intel.com](mailto:ashok.raj@intel.com)>, Joel Schopp <[jschopp@austin.ibm.com](mailto:jschopp@austin.ibm.com)>

### 简介

现代系统架构的演进已经在处理器中引入了先进的错误报告和纠正能力。有一些 OEM 也支持可热拔插的 NUMA (Non Uniform Memory Access, 非统一内存访问) 硬件, 其中物理节点的插入和移除需要支持 CPU 热插拔。

这样的进步要求内核可用的 CPU 被移除, 要么是出于配置的原因, 要么是出于 RAS 的目的, 以保持一个不需要的 CPU 不在系统执行路径。因此需要在 Linux 内核中支持 CPU 热拔插。

CPU 热拔插支持的一个更新颖的用途是它在 SMP 的暂停恢复支持中的应用。双核和超线程支持使得即使是笔记本电脑也能运行不支持这些方法的 SMP 内核。

### 命令行开关

**maxcpus=n** 限制启动时的 CPU 为  $n$  个。例如, 如果你有四个 CPU, 使用 `maxcpus=2` 将只能启动两个。你可以选择稍后让其他 CPU 上线。

**nr\_cpus=n** 限制内核将支持的 CPU 总量。如果这里提供的数量低于实际可用的 CPU 数量, 那么其他 CPU 以后就不能上线了。

**additional\_cpus=n** 使用它来限制可热插拔的 CPU。该选项设置 `cpu_possible_mask = cpu_present_mask + additional_cpus`

这个选项只限于 IA64 架构。

**possible\_cpus=n** 这个选项设置 `cpu_possible_mask` 中的 `possible_cpus` 位。

这个选项只限于 X86 和 S390 架构。

**cpu0\_hotplug** 允许关闭 CPU0。

这个选项只限于 X86 架构。

## CPU 位图

**cpu\_possible\_mask** 系统中可能可用 CPU 的位图。这是用来为 per\_cpu 变量分配一些启动时的内存，这些变量不会随着 CPU 的可用或移除而增加/减少。一旦在启动时的发现阶段被设置，该映射就是静态的，也就是说，任何时候都不会增加或删除任何位。根据你的系统需求提前准确地调整它可以节省一些启动时的内存。

**cpu\_online\_mask** 当前在线的所有 CPU 的位图。在一个 CPU 可用于内核调度并准备接收设备的中断后，它被设置在 \_\_cpu\_up() 中。当使用 \_\_cpu\_disable() 关闭一个 CPU 时，它被清空，在此之前，所有的操作系统服务包括中断都被迁移到另一个目标 CPU。

**cpu\_present\_mask** 系统中当前存在的 CPU 的位图。它们并非全部在线。当物理热拔插被相关的子系统（如 ACPI）处理时，可以改变和添加新的位或从位图中删除，这取决于事件是 hot-add/hot-remove。目前还没有定死规定。典型的用法是在启动时启动拓扑结构，这时热插拔被禁用。

你真的不需要操作任何系统的 CPU 映射。在大多数情况下，它们应该是只读的。当设置每个 CPU 资源时，几乎总是使用 cpu\_possible\_mask 或 for\_each\_possible\_cpu() 来进行迭代。宏 for\_each\_cpu() 可以用来迭代一个自定义的 CPU 掩码。

不要使用 cpumask\_t 以外的任何东西来表示 CPU 的位图。

## 使用 CPU 热拔插

内核选项 CONFIG\_HOTPLUG\_CPU 需要被启用。它目前可用于多种架构，包括 ARM、MIPS、PowerPC 和 X86。配置是通过 sysfs 接口完成的：

```
$ ls -lh /sys/devices/system/cpu
total 0
drwxr-xr-x 9 root root 0 Dec 21 16:33 cpu0
drwxr-xr-x 9 root root 0 Dec 21 16:33 cpu1
drwxr-xr-x 9 root root 0 Dec 21 16:33 cpu2
drwxr-xr-x 9 root root 0 Dec 21 16:33 cpu3
drwxr-xr-x 9 root root 0 Dec 21 16:33 cpu4
drwxr-xr-x 9 root root 0 Dec 21 16:33 cpu5
drwxr-xr-x 9 root root 0 Dec 21 16:33 cpu6
drwxr-xr-x 9 root root 0 Dec 21 16:33 cpu7
drwxr-xr-x 2 root root 0 Dec 21 16:33 hotplug
-r--r--r-- 1 root root 4.0K Dec 21 16:33 offline
-r--r--r-- 1 root root 4.0K Dec 21 16:33 online
-r--r--r-- 1 root root 4.0K Dec 21 16:33 possible
-r--r--r-- 1 root root 4.0K Dec 21 16:33 present
```

文件 *offline*、*online*、*possible*、*present* 代表 CPU 掩码。每个 CPU 文件夹包含一个 *online* 文件，控制逻辑上的开（1）和关（0）状态。要在逻辑上关闭 CPU4：

```
$ echo 0 > /sys/devices/system/cpu/cpu4/online
smpboot: CPU 4 is now offline
```

一旦 CPU 被关闭，它将从 `/proc/interrupts`、`/proc/cpuinfo` 中被删除，也不应该被 `top` 命令显示出来。要让 CPU4 重新上线：

```
$ echo 1 > /sys/devices/system/cpu/cpu4/online
smpboot: Booting Node 0 Processor 4 APIC 0x1
```

CPU 又可以使用了。这应该对所有的 CPU 都有效。CPU0 通常比较特殊，被排除在 CPU 热拔插之外。在 X86 上，内核选项 `CONFIG_BOOTPARAM_HOTPLUG_CPU0` 必须被启用，以便能够关闭 CPU0。或者，可以使用内核命令选项 `cpu0_hotplug`。CPU0 的一些已知的依赖性：

- 从休眠/暂停中恢复。如果 CPU0 处于离线状态，休眠/暂停将失败。
- PIC 中断。如果检测到 PIC 中断，CPU0 就不能被移除。

如果你发现 CPU0 上有任何依赖性，请告知 Fenghua Yu <[fenghua.yu@intel.com](mailto:fenghua.yu@intel.com)>。

## CPU 的热拔插协作

### 下线情况

一旦 CPU 被逻辑关闭，注册的热插拔状态的清除回调将被调用，从 `CPUHP_ONLINE` 开始，在 `CPUHP_OFFLINE` 状态结束。这包括：

- 如果任务因暂停操作而被冻结，那么 `cpuhp_tasks_frozen` 将被设置为 `true`。
- 所有进程都会从这个将要离线的 CPU 迁移到新的 CPU 上。新的 CPU 是从每个进程的当前 cpuset 中选择的，它可能是所有在线 CPU 的一个子集。
- 所有针对这个 CPU 的中断都被迁移到新的 CPU 上。
- 计时器也会被迁移到新的 CPU 上。
- 一旦所有的服务被迁移，内核会调用一个特定的例程 `_cpu_disable()` 来进行特定的清理。

## 使用热插拔 API

一旦一个 CPU 下线或上线，就有可能收到通知。这对某些需要根据可用 CPU 数量执行某种设置或清理功能的驱动程序来说可能很重要：

```
#include <linux/cpuhotplug.h>

ret = cpuhp_setup_state(CPUHP_AP_ONLINE_DYN, "X/Y:online",
                        Y_online, Y_prepare_down);
```

`X` 是子系统，`Y` 是特定的驱动程序。`Y_online` 回调将在所有在线 CPU 的注册过程中被调用。如果在线回调期间发生错误，`Y_prepare_down` 回调将在所有之前调用过在线回调的 CPU 上调用。注册完成后，一旦有 CPU 上线，`Y_online` 回调将被调用，当 CPU 关闭时，`Y_prepare_down` 将被调用。所有之前在 `Y_online` 中分配的资源都应该在 `Y_prepare_down` 中释放。如果在注册过程中发生错误，返回值 `ret` 为负值。否则会

返回一个正值，其中包含动态分配状态 (`CPUHP_AP_ONLINE_DYN`) 的分配热拔插。对于预定义的状态，它将返回 0。

该回调可以通过调用 `cpuhp_remove_state()` 来删除。如果是动态分配的状态 (`CPUHP_AP_ONLINE_DYN`)，则使用返回的状态。在移除热插拔状态的过程中，将调用拆解回调。

## 多个实例

如果一个驱动程序有多个实例，并且每个实例都需要独立执行回调，那么很可能应该使用 `multi-state`。首先需要注册一个多状态的状态：

```
ret = cphhp_setup_state_multi(CPUHP_AP_ONLINE_DYN, "X/Y:online",
                               Y_online, Y_prepare_down);
Y_hp_online = ret;
```

`cpuhp_setup_state_multi()` 的行为与 `cpuhp_setup_state()` 类似，只是它为多状态准备了回调，但不调用回调。这是一个一次性的设置。一旦分配了一个新的实例，你需要注册这个新实例：

```
ret = cphhp_state_add_instance(Y_hp_online, &d->node);
```

这个函数将把这个实例添加到你先前分配的 `Y_hp_online` 状态，并在所有在线的 CPU 上调用先前注册的回调 (`Y_online`)。`node` 元素是你的每个实例数据结构中的一个 `struct hlist_node` 成员。

在移除该实例时：

```
cpuhp_state_remove_instance(Y_hp_online, &d->node)
```

应该被调用，这将在所有在线 CPU 上调用拆分回调。

## 手动设置

通常情况下，在注册或移除状态时调用 `setup` 和 `teardown` 回调是很方便的，因为通常在 CPU 上线（下线）和驱动的初始设置（关闭）时需要执行该操作。然而，每个注册和删除功能也有一个 `_nocalls` 的后缀，如果不希望调用回调，则不调用所提供的回调。在手动设置（或关闭）期间，应该使用 `get_online_cpus()` 和 `put_online_cpus()` 函数来抑制 CPU 热插拔操作。

## 事件的顺序

热插拔状态被定义在 `include/linux/cpuhotplug.h`:

- `CPUHP_OFFLINE` … `CPUHP_AP_OFFLINE` 状态是在 CPU 启动前调用的。
- `CPUHP_AP_OFFLINE` … `CPUHP_AP_ONLINE` 状态是在 CPU 被启动后被调用的。中断是关闭的，调度程序还没有在这个 CPU 上活动。从 `CPUHP_AP_OFFLINE` 开始，回调被调用到目标 CPU 上。
- `CPUHP_AP_ONLINE_DYN` 和 `CPUHP_AP_ONLINE_DYN_END` 之间的状态被保留给动态分配。

- 这些状态在 CPU 关闭时以相反的顺序调用，从 `CPUHP_ONLINE` 开始，在 `CPUHP_OFFLINE` 停止。这里的回调是在将被关闭的 CPU 上调用的，直到 `CPUHP_AP_OFFLINE`。

通过 `CPUHP_AP_ONLINE_DYN` 动态分配的状态通常已经足够了。然而，如果在启动或关闭期间需要更早的调用，那么应该获得一个显式状态。如果热拔插事件需要相对于另一个热拔插事件的特定排序，也可能需要一个显式状态。

### 测试热拔插状态

验证自定义状态是否按预期工作的一个方法是关闭一个 CPU，然后再把它上线。也可以把 CPU 放到某些状态（例如 `CPUHP_AP_ONLINE`），然后再回到 `CPUHP_ONLINE`。这将模拟在 `CPUHP_AP_ONLINE` 之后的一个状态出现错误，从而导致回滚到在线状态。

所有注册的状态都被列举在 `/sys/devices/system/cpu/hotplug/states`

```
$ tail /sys/devices/system/cpu/hotplug/states
138: mm/vmscan:online
139: mm/vmstat:online
140: lib/percpu_cnt:online
141: acpi/cpu-drv:online
142: base/cacheinfo:online
143: virtio/net:online
144: x86/mce:online
145: printk:online
168: sched:active
169: online
```

要将 CPU4 回滚到 `lib/percpu_cnt:online`，再回到在线状态，只需发出：

```
$ cat /sys/devices/system/cpu/cpu4/hotplug/state
169
$ echo 140 > /sys/devices/system/cpu/cpu4/hotplug/target
$ cat /sys/devices/system/cpu/cpu4/hotplug/state
140
```

需要注意的是，状态 140 的清除回调已经被调用。现在重新上线：

```
$ echo 169 > /sys/devices/system/cpu/cpu4/hotplug/target
$ cat /sys/devices/system/cpu/cpu4/hotplug/state
169
```

启用追踪事件后，单个步骤也是可见的：

```
#  TASK-PID    CPU#    TIMESTAMP   FUNCTION
#    | |        |        |
bash-394  [001]  22.976: cpuhp_enter: cpu: 0004 target: 140 step: 169 (cpuhp_kick_ap_work)
cpuhp/4-31  [004]  22.977: cpuhp_enter: cpu: 0004 target: 140 step: 168 (sched_cpu_
↳deactivate)
cpuhp/4-31  [004]  22.990: cpuhp_exit:  cpu: 0004 state: 168 step: 168 ret: 0
```

```

cpuhp/4-31 [004] 22.991: cphu_hp_enter: cpu: 0004 target: 140 step: 144 (mce_cpu_pre_down)
cpuhp/4-31 [004] 22.992: cphu_hp_exit: cpu: 0004 state: 144 step: 144 ret: 0
cpuhp/4-31 [004] 22.993: cphu_hp_multi_enter: cpu: 0004 target: 140 step: 143 (virtnet_cpu_
↪down_prep)
cpuhp/4-31 [004] 22.994: cphu_hp_exit: cpu: 0004 state: 143 step: 143 ret: 0
cpuhp/4-31 [004] 22.995: cphu_hp_enter: cpu: 0004 target: 140 step: 142 (cacheinfo_cpu_pre_
↪down)
cpuhp/4-31 [004] 22.996: cphu_hp_exit: cpu: 0004 state: 142 step: 142 ret: 0
  bash-394 [001] 22.997: cphu_hp_exit: cpu: 0004 state: 140 step: 169 ret: 0
  bash-394 [005] 95.540: cphu_hp_enter: cpu: 0004 target: 169 step: 140 (cpuhp_kick_ap_work)
cpuhp/4-31 [004] 95.541: cphu_hp_enter: cpu: 0004 target: 169 step: 141 (acpi_soft_cpu_
↪online)
cpuhp/4-31 [004] 95.542: cphu_hp_exit: cpu: 0004 state: 141 step: 141 ret: 0
cpuhp/4-31 [004] 95.543: cphu_hp_enter: cpu: 0004 target: 169 step: 142 (cacheinfo_cpu_
↪online)
cpuhp/4-31 [004] 95.544: cphu_hp_exit: cpu: 0004 state: 142 step: 142 ret: 0
cpuhp/4-31 [004] 95.545: cphu_hp_multi_enter: cpu: 0004 target: 169 step: 143 (virtnet_cpu_
↪online)
cpuhp/4-31 [004] 95.546: cphu_hp_exit: cpu: 0004 state: 143 step: 143 ret: 0
cpuhp/4-31 [004] 95.547: cphu_hp_enter: cpu: 0004 target: 169 step: 144 (mce_cpu_online)
cpuhp/4-31 [004] 95.548: cphu_hp_exit: cpu: 0004 state: 144 step: 144 ret: 0
cpuhp/4-31 [004] 95.549: cphu_hp_enter: cpu: 0004 target: 169 step: 145 (console_cpu_notify)
cpuhp/4-31 [004] 95.550: cphu_hp_exit: cpu: 0004 state: 145 step: 145 ret: 0
cpuhp/4-31 [004] 95.551: cphu_hp_enter: cpu: 0004 target: 169 step: 168 (sched_cpu_activate)
cpuhp/4-31 [004] 95.552: cphu_hp_exit: cpu: 0004 state: 168 step: 168 ret: 0
  bash-394 [005] 95.553: cphu_hp_exit: cpu: 0004 state: 169 step: 140 ret: 0

```

可以看到，CPU4 一直下降到时间戳 22.996，然后又上升到 95.552。所有被调用的回调，包括它们的返回代码都可以在跟踪中看到。

## 架构的要求

需要具备以下功能和配置：

**CONFIG\_HOTPLUG\_CPU** 这个配置项需要在 Kconfig 中启用

**\_cpu\_up()** 调出一个 cpu 的架构接口

**\_cpu\_disable()** 关闭 CPU 的架构接口，在此程序返回后，内核不能再处理任何中断。这包括定时器的关闭。

**\_cpu\_die()** 这实际上是为了确保 CPU 的死亡。实际上，看看其他架构中实现 CPU 热拔插的一些示例代码。

对于那个特定的架构，处理器被从 **idle()** 循环中拿下来。**\_cpu\_die()** 通常会等待一些 **per\_cpu** 状态的设置，以确保处理器的死亡例程被调用来保持活跃。

### 用户空间通知

在 CPU 成功上线或下线后，udev 事件被发送。一个 udev 规则，比如：

```
SUBSYSTEM=="cpu", DRIVERS=="processor", DEVPATH=="/devices/system/cpu/*", RUN+="the_hotplug_"
˓→receiver.sh"
```

将接收所有事件。一个像这样的脚本：

```
#!/bin/sh

if [ "${ACTION}" = "offline" ]
then
    echo "CPU ${DEVPATH##*/} offline"

elif [ "${ACTION}" = "online" ]
then
    echo "CPU ${DEVPATH##*/} online"

fi
```

可以进一步处理该事件。

### 内核内联文档参考

该 API 在以下内核代码中：

include/linux/cpuhotplug.h

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

---

**Original** Documentation/core-api/genericirq.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 吴想成 Wu XiangCheng <[bowwxc@email.cn](mailto:bowwxc@email.cn)>

## Linux 通用 IRQ 处理

版权 © 2005-2010: Thomas Gleixner

版权 © 2005-2006: Ingo Molnar

## 简介

通用中断处理层是为了给设备驱动程序提供一个完整的中断处理抽象（层）。它能够处理所有不同类型的中断控制器硬件。设备驱动程序使用通用 API 函数来请求、启用、禁用和释放中断。驱动程序不需要知道任何关于硬件处理中断的细节，所以它们可以在不同的平台上使用而不需要修改代码。

本文档提供给那些希望在通用 IRQ 处理层的帮助下实现基于其架构的中断子系统的开发者。

## 理论依据

Linux 中中断处理的原始实现使用 `_do_IRQ()` 超级处理器，它能够处理每种类型的中断逻辑。

最初，Russell King 确定了不同类型的处理器，以便为 Linux 2.5/2.6 中的 ARM 中断处理程序实现建立一个相当通用的集合。他区分了以下几种类型：

- 电平触发型
- 边沿触发型
- 简单型

在实现过程中，我们发现了另一种类型：

- 响应 EOI (end of interrupt) 型

在 SMP 的 `_do_IRQ()` 超级处理器中，还需定义一种类型：

- 每 cpu 型 (针对 CPU SMP)

这种高层 IRQ 处理程序的拆分实现使我们能够为每个特定的中断类型优化中断处理的流程。这减少了该特定代码路径的复杂性，并允许对特定类型进行优化处理。

最初的通用 IRQ 实现使用 `hw_interrupt_type` 结构体及其 `->ack` `->end` 等回调来区分超级处理器中的流控制。这导致了流逻辑和低级硬件逻辑的混合，也导致了不必要的代码重复：例如 i386 中的 `ioapic_level_irq` 和 `ioapic_edge_irq`，这两个 IRQ 类型共享许多低级的细节，但有不同的流处理。

一个更自然的抽象是“irq 流”和“芯片细节”的干净分离。

分析一些架构的 IRQ 子系统的实现可以发现，他们中的大多数可以使用一套通用的“irq 流”方法，只需要添加芯片级的特定代码。这种分离对于那些需要 IRQ 流本身而不需要芯片细节的特定（子）架构也很有价值——以提供了一个更透明的 IRQ 子系统设计。

每个中断描述符都被分配给它自己的高层流程处理器，这通常是一个通用的实现。（这种高层次的流程处理器的实现也使得提供解复用处理器变得简单，这可以在各种架构的嵌入式平台上找到。）

这种分离使得通用中断处理层更加灵活和可扩展。例如，一个（子）架构可以使用通用的 IRQ 流实现“电平触发型”中断，并添加一个（子）架构特定的“边沿型”实现。

为了使向新模型的过渡更容易，并防止破坏现有实现，`_do_IRQ()` 超级处理程序仍然可用。这导致了一种暂时的双重性。随着时间的推移，新的模型应该在越来越多的架构中被使用，因为它能使 IRQ 子系统更小更干净。它已经被废弃三年了，即将被删除。

### 已知的缺陷和假设

没有（但愿如此）。

### 抽象层

中断代码中主要有三个抽象层次：

1. 高级别的驱动 API
2. 高级别的 IRQ 流处理器
3. 芯片级的硬件封装

### 中断控制流

每个中断都由一个中断描述符结构体 `irq_desc` 来描述。中断是由一个“无符号整型”的数值来引用的，它在描述符结构体数组中选择相应的中断描述符结构体。描述符结构体包含状态信息和指向中断流方法和中断芯片结构的指针，这些都是分配给这个中断的。

每当中断触发时，低级架构代码通过调用 `desc->handle_irq()` 调用到通用中断代码中。这个高层 IRQ 处理函数只使用由分配的芯片描述符结构体引用的 `desc->irq_data.chip` 基元。

### 高级驱动程序 API

高层驱动 API 由以下函数组成：

- `request_irq()`
- `request_threaded_irq()`
- `free_irq()`
- `disable_irq()`
- `enable_irq()`
- `disable_irq_nosync()` (SMP only)
- `synchronize_irq()` (SMP only)

- irq\_set\_irq\_type()
- irq\_set\_irq\_wake()
- irq\_set\_handler\_data()
- irq\_set\_chip()
- irq\_set\_chip\_data()

详见自动生成的函数文档。

**Note:** 由于文档构建流程所限，中文文档中并没有引入自动生成的函数文档，所以请读者直接阅读源码注释。

## 电平触发型 IRQ 流处理程序

通用层提供了一套预定义的 irq-flow 方法：

- handle\_level\_irq()
- handle\_edge\_irq()
- handle\_fasteoi\_irq()
- handle\_simple\_irq()
- handle\_percpu\_irq()
- handle\_edge\_eoi\_irq()
- handle\_bad\_irq()

中断流处理程序（无论是预定义的还是架构特定的）由架构在启动期间或设备初始化期间分配给特定中断。

## 默认流实现

### 辅助函数

辅助函数调用芯片基元，并被默认流实现所使用。以下是实现的辅助函数（简化摘录）：

```
default_enable(struct irq_data *data)
{
    desc->irq_data(chip->irq_unmask(data));
}

default_disable(struct irq_data *data)
{
    if (!delay_disable(data))
        desc->irq_data(chip->irq_mask(data));
```

```
}
```

```
default_ack(struct irq_data *data)
{
    chip->irq_ack(data);
}
```

```
default_mask_ack(struct irq_data *data)
{
    if (chip->irq_mask_ack) {
        chip->irq_mask_ack(data);
    } else {
        chip->irq_mask(data);
        chip->irq_ack(data);
    }
}
```

```
noop(struct irq_data *data)
{}
```

### 默认流处理程序的实现

#### 电平触发型 IRQ 流处理器

`handle_level_irq` 为电平触发型的中断提供了一个通用实现。

实现的控制流如下（简化摘录）：

```
desc->irq_data(chip->irq_mask_ack());
handle_irq_event(desc->action);
desc->irq_data(chip->irq_unmask());
```

#### 默认的需回应 IRQ 流处理器

`handle_fasteoi_irq` 为中断提供了一个通用的实现，它只需要在处理程序的末端有一个 EOI。

实现的控制流如下（简化摘录）：

```
handle_irq_event(desc->action);
desc->irq_data(chip->irq_eoi());
```

## 默认的边沿触发型 IRQ 流处理器

`handle_edge_irq` 为边沿触发型的中断提供了一个通用的实现。

实现的控制流如下（简化摘录）：

```
if (desc->status & running) {
    desc->irq_data(chip->irq_mask_ack());
    desc->status |= pending | masked;
    return;
}
desc->irq_data(chip->irq_ack());
desc->status |= running;
do {
    if (desc->status & masked)
        desc->irq_data(chip->irq_unmask());
    desc->status &= ~pending;
    handle_irq_event(desc->action);
} while (status & pending);
desc->status &= ~running;
```

## 默认的简单型 IRQ 流处理器

`handle_simple_irq` 提供了一个简单型中断的通用实现。

---

**Note:** 简单型的流处理器不调用任何处理程序/芯片基元。

---

实现的控制流程如下（简化摘录）：

```
handle_irq_event(desc->action);
```

## 默认的每 CPU 型流处理器程序

`handle_percpu_irq` 为每 CPU 型中断提供一个通用的实现。

每个 CPU 中断只在 SMP 上可用，该处理程序提供了一个没有锁的简化版本。

以下是控制流的实现（简化摘录）：

```
if (desc->irq_data(chip->irq_ack))
    desc->irq_data(chip->irq_ack());
handle_irq_event(desc->action);
if (desc->irq_data(chip->irq_eoi))
    desc->irq_data(chip->irq_eoi());
```

### EOI 边沿型 IRQ 流处理器

`handle_edge_eoi_irq` 提供了一个异常的边沿触发型处理程序，它只用于拯救 powerpc/cell 上的一个严重失控的 irq 控制器。

### 坏的 IRQ 流处理器

`handle_bad_irq` 用于处理没有真正分配处理程序的假中断。

### 特殊性和优化

通用函数是为“干净”的架构和芯片设计的，它们没有平台特定的 IRQ 处理特殊性。如果一个架构需要在“流”的层面上实现特殊性，那么它可以通过覆盖高层的 IRQ-流处理程序来实现。

### 延迟中断禁用

每个中断可选择的功能是由 Russell King 在 ARM 中断实现中引入的，当调用 `disable_irq()` 时，不会在硬件层面上屏蔽中断。中断保持启用状态，而在中断事件发生时在流处理器中被屏蔽。这可以防止在硬件上丢失边沿中断，因为硬件上不存储边沿中断事件，而中断在硬件级被禁用。当一个中断在 `IRQ_DISABLED` 标志被设置时到达，那么该中断在硬件层面被屏蔽，`IRQ_PENDING` 位被设置。当中断被 `enable_irq()` 重新启用时，将检查挂起位，如果它被设置，中断将通过硬件或软件重发机制重新发送。(当你想使用延迟中断禁用功能，而你的硬件又不能重新触发中断时，有必要启用 `CONFIG_HARDIRQS_SW_RESEND`。) 延迟中断禁止功能是不可配置的。

### 芯片级硬件封装

芯片级硬件描述符结构体 `irq_chip` 包含了所有与芯片直接相关的功能，这些功能可以被 irq 流实现所利用。

- `irq_ack`
- `irq_mask_ack` - 可选的，建议使用的性能
- `irq_mask`
- `irq_unmask`
- `irq_eoi` - 可选的，EOI 流处理程序需要
- `irq_retrigger` - 可选的
- `irq_set_type` - 可选的
- `irq_set_wake` - 可选的

这些基元的意思是严格意义上的：`ack` 是指 ACK，`masking` 是指对 IRQ 线的屏蔽，等等。这取决于流处理器如何使用这些基本的低级功能单元。

## \_\_do\_IRQ 入口点

最初的实现 `__do_IRQ()` 是所有类型中断的替代入口点。它已经不存在了。

这个处理程序被证明不适合所有的中断硬件，因此被重新实现了边沿/级别/简单/超高速中断的拆分功能。这不仅是一个功能优化。它也缩短了中断的代码路径。

## 在 SMP 上的锁

芯片寄存器的锁定是由定义芯片基元的架构决定的。每个寄存器的结构通过 `desc->lock`, 由通用层保护。

## 通用中断芯片

为了避免复制相同的 IRQ 芯片实现，核心提供了一个可配置的通用中断芯片实现。开发者在自己实现相同的功能之前，应该仔细检查通用芯片是否符合他们的需求，并以稍微不同的方式实现相同的功能。

该 API 在以下内核代码中：

`kernel/irq/generic-chip.c`

## 结构体

本章包含自动生成的结构体文档，这些结构体在通用 IRQ 层中使用。

该 API 在以下内核代码中：

`include/linux/irq.h`

`include/linux/interrupt.h`

## 提供的通用函数

这一章包含了自动生成的内核 API 函数的文档，这些函数被导出。

该 API 在以下内核代码中：

`kernel/irq/manage.c`

`kernel/irq/chip.c`

### 提供的内部函数

本章包含自动生成的内部函数的文档。

该 API 在以下内核代码中:

kernel/irq/irqdesc.c

kernel/irq/handle.c

kernel/irq/chip.c

### 鸣谢

感谢以下人士对本文档作出的贡献:

1. Thomas Gleixner [tglx@linutronix.de](mailto:tglx@linutronix.de)
2. Ingo Molnar [mingo@elte.hu](mailto:mingo@elte.hu)

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/core-api/memory-hotplug.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

## 内存热插拔

### 内存热拔插事件通知器

热插拔事件被发送到一个通知队列中。

在 `include/linux/memory.h` 中定义了六种类型的通知:

**MEM\_GOING\_ONLINE** 在新内存可用之前生成，以便能够为子系统处理内存做准备。页面分配器仍然无法从新的内存中进行分配。

**MEM\_CANCEL\_ONLINE** 如果 **MEM\_GOING\_ONLINE** 失败，则生成。

**MEM\_ONLINE** 当内存成功上线时产生。回调可以从新的内存中分配页面。

**MEM\_GOING\_OFFLINE** 在开始对内存进行下线处理时生成。从内存中的分配不再可能，但是一些要下线的内存仍然在使用。回调可以用来释放一个子系统在指定内存块中已知的内存。

**MEM\_CANCEL\_OFFLINE** 如果 MEM\_GOING\_OFFLINE 失败，则生成。来自我们试图离线的内存块中的内存又可以使用了。

**MEM\_OFFLINE** 在内存下线完成后生成。

可以通过调用如下函数来注册一个回调程序：

```
hotplug_memory_notifier(callback_func, priority)
```

优先级数值较高的回调函数在数值较低的回调函数之前被调用。

一个回调函数必须有以下原型：

```
int callback_func(
    struct notifier_block *self, unsigned long action, void *arg);
```

回调函数的第一个参数 (self) 是指向回调函数本身的通知器链块的一个指针。第二个参数 (action) 是上述的事件类型之一。第三个参数 (arg) 传递一个指向 memory\_notify 结构体的指针：

```
struct memory_notify {
    unsigned long start_pfn;
    unsigned long nr_pages;
    int status_change_nid_normal;
    int status_change_nid;
}
```

- start\_pfn 是在线/离线内存的 start\_pfn。
- nr\_pages 是在线/离线内存的页数。
- status\_change\_nid\_normal 是当 nodemask 的 N\_NORMAL\_MEMORY 被设置/清除时设置节点 id，如果是-1，则 nodemask 状态不改变。
- status\_change\_nid 是当 nodemask 的 N\_MEMORY 被（将）设置/清除时设置的节点 id。这意味着一个新的（没上线的）节点通过联机获得新的内存，而一个节点失去了所有的内存。如果这个值为-1，那么 nodemask 的状态就不会改变。

如果 status\_change\_nid\* >= 0，回调应该在必要时为节点创建/丢弃结构体。

回调程序应返回 include/linux/notifier.h 中定义的 NOTIFY\_DONE, NOTIFY\_OK, NOTIFY\_BAD, NOTIFY\_STOP 中的一个值。

NOTIFY\_DONE 和 NOTIFY\_OK 对进一步处理没有影响。

NOTIFY\_BAD 是作为对 MEM\_GOING\_ONLINE、MEM\_GOING\_OFFLINE、MEM\_ONLINE 或 MEM\_OFFLINE 动作的回应，用于取消热插拔。它停止对通知队列的进一步处理。

NOTIFY\_STOP 停止对通知队列的进一步处理。

### 内部锁

当添加/删除使用内存块设备（即普通 RAM）的内存时，`device_hotplug_lock` 应该被保持为：

- 针对在线/离线请求进行同步（例如，通过 sysfs）。这样一来，内存块设备只有在内存被完全添加后才能被用户空间访问（`.online/.state` 属性）。而在删除内存时，我们知道没有人在临界区。
- 与 CPU 热拔插或类似操作同步（例如 ACPI 和 PPC 相关操作）

特别是，在添加内存和用户空间试图以比预期更快的速度上线该内存时，有可能出现锁反转，使用 `device_hotplug_lock` 可以避免此情况：

- `device_online()` 将首先接受 `device_lock()`，然后是 `mem_hotplug_lock`。
- `add_memory_resource()` 将首先使用 `mem_hotplug_lock`，然后是 `device_lock()`（在创建设备时，在 `bus_add_device()` 期间）。

由于在使用 `device_lock()` 之前，设备对用户空间是可见的，这可能导致锁的反转。

内存的上线/下线应该通过 `device_online()/device_offline()` 完成——确保它与通过 sysfs 进行的操作正确同步。建议持有 `device_hotplug_lock`（例如，保护 `online_type`）。

当添加/删除/上线/下线内存或者添加/删除异构或设备内存时，我们应该始终持有写模式的 `mem_hotplug_lock`，以序列化内存热插拔（例如访问全局/区域变量）。

此外，`mem_hotplug_lock`（与 `device_hotplug_lock` 相反）在读取模式下允许一个相当有效的 `get_online_mems/put_online_mems` 实现，所以访问内存的代码可以防止该内存消失。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:<alexs@kernel.org>)。

---

**Original** Documentation/core-api/protection-keys.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 吴想成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)>

## 内存保护密钥

用户空间的内存保护密钥（Memory Protection Keys for Userspace, PKU, 亦即 PKEYs）是英特尔 Skylake（及以后）“可扩展处理器”服务器 CPU 上的一项功能。它将在未来的非服务器英特尔处理器和未来的 AMD 处理器中可用。

对于任何希望测试或使用该功能的人来说，它在亚马逊的 EC2 C5 实例中是可用的，并且已知可以在那里使用 Ubuntu 17.04 镜像运行。

内存保护密钥提供了一种机制来执行基于页面的保护，但在应用程序改变保护域时不需要修改页表。它的工作原理是在每个页表项中为“保护密钥”分配 4 个以前被忽略的位，从而提供 16 个可能的密钥。

还有一个新的用户可访问寄存器（PKRU），为每个密钥提供两个单独的位（访问禁止和写入禁止）。作为一个 CPU 寄存器，PKRU 在本质上是线程本地的，可能会给每个线程提供一套不同于其他线程的保护措施。

有两条新指令（RDPKRU/WRPKRU）用于读取和写入新的寄存器。该功能仅在 64 位模式下可用，尽管物理地址扩展页表中理论上有空间。这些权限只在数据访问上强制执行，对指令获取没有影响。

## 系统调用

有 3 个系统调用可以直接与 pkeys 进行交互：

```
int pkey_alloc(unsigned long flags, unsigned long init_access_rights)
int pkey_free(int pkey);
int pkey_mprotect(unsigned long start, size_t len,
                  unsigned long prot, int pkey);
```

在使用一个 pkey 之前，必须先用 pkey\_alloc() 分配它。一个应用程序直接调用 WRPKRU 指令，以改变一个密钥覆盖的内存的访问权限。在这个例子中，WRPKRU 被一个叫做 pkey\_set() 的 C 函数所封装：

```
int real_prot = PROT_READ|PROT_WRITE;
pkey = pkey_alloc(0, PKEY_DISABLE_WRITE);
ptr = mmap(NULL, PAGE_SIZE, PROT_NONE, MAP_ANONYMOUS|MAP_PRIVATE, -1, 0);
ret = pkey_mprotect(ptr, PAGE_SIZE, real_prot, pkey);
... application runs here
```

现在，如果应用程序需要更新‘ptr’处的数据，它可以获取访问权，进行更新，然后取消其写访问权：

```
pkey_set(pkey, 0); // clear PKEY_DISABLE_WRITE
*ptr = foo; // assign something
pkey_set(pkey, PKEY_DISABLE_WRITE); // set PKEY_DISABLE_WRITE again
```

现在，当它释放内存时，它也将释放 pkey，因为它不再被使用了：

```
munmap(ptr, PAGE_SIZE);
pkey_free(pkey);
```

**Note:** pkey\_set() 是 RDPKRU 和 WRPKRU 指令的一个封装器。在 tools/testing/selftests/x86/protection\_keys.c 中可以找到一个实现实例。 tools/testing/selftests/x86/protection\_keys.c.

---

## 行为

内核试图使保护密钥与普通的 mprotect() 的行为一致。例如，如果你这样做：

```
mprotect(ptr, size, PROT_NONE);  
something(ptr);
```

这样做的时候，你可以期待保护密钥的相同效果：

```
pkey = pkey_alloc(0, PKEY_DISABLE_WRITE | PKEY_DISABLE_READ);  
pkey_mprotect(ptr, size, PROT_READ|PROT_WRITE, pkey);  
something(ptr);
```

无论 something() 是否是对'ptr'的直接访问，这都应该为真。如：

```
*ptr = foo;
```

或者当内核代表应用程序进行访问时，比如 read()：

```
read(fd, ptr, 1);
```

在这两种情况下，内核都会发送一个 SIGSEGV，但当违反保护密钥时，si\_code 将被设置为 SEGV\_PKERR，而当违反普通的 mprotect() 权限时，则是 SEGV\_ACCERR。

Todolist:

```
memory-hotplug cpu_hotplug genericirq
```

## 内存管理

如何在内核中分配和使用内存。请注意，在 /vm/index 中有更多的内存管理文档。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/core-api/memory-allocation.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 时奎亮 <[alexs@kernel.org](mailto:alexs@kernel.org)>

## 内存分配指南

Linux 为内存分配提供了多种 API。你可以使用 *kmalloc* 或 *kmem\_cache\_alloc* 系列分配小块内存，使用 *vmalloc* 及其派生产品分配大的几乎连续的区域，或者你可以用 *alloc\_pages* 直接向页面分配器请求页面。也可以使用更专业的分配器，例如 *cma\_alloc* 或 *zs\_malloc*。

大多数的内存分配 API 使用 GFP 标志来表达该内存应该如何分配。GFP 的缩写代表“(get free pages) 获取空闲页”，是底层的内存分配功能。

(内存) 分配 API 的多样性与众多的 GFP 标志相结合，使得“我应该如何分配内存？”这个问题不那么容易回答，尽管很可能你应该使用

```
kzalloc(<size>, GFP_KERNEL);
```

当然，有些情况下必须使用其他分配 API 和不同的 GFP 标志。

## 获取空闲页标志

GFP 标志控制分配器的行为。它们告诉我们哪些内存区域可以被使用，分配器应该多努力寻找空闲的内存，这些内存是否可以被用户空间访问等等。内存管理 API 为 GFP 标志和它们的组合提供了参考文件，这里我们简要介绍一下它们的推荐用法：

- 大多数时候，`GFP_KERNEL` 是你需要的。内核数据结构的内存，DMA 可用内存，inode 缓存，所有这些和其他许多分配类型都可以使用 `GFP_KERNEL`。注意，使用 `GFP_KERNEL` 意味着 `GFP_RECLAIM`，这意味着在有内存压力的情况下可能会触发直接回收；调用上下文必须允许睡眠。
- 如果分配是从一个原子上下文中进行的，例如中断处理程序，使用 `GFP_NOWAIT`。这个标志可以防止直接回收和 IO 或文件系统操作。因此，在内存压力下，`GFP_NOWAIT` 分配可能会失败。有合理退路的分配应该使用 `GFP_NOWARN`。
- 如果你认为访问保留内存区是合理的，并且除非分配成功，否则内核会有压力，你可以使用 `GFP_ATOMIC`。
- 从用户空间触发的不可信任的分配应该是 `kmem` 核算的对象，必须设置 `__GFP_ACCOUNT` 位。有一个方便的用于 `GFP_KERNEL` 分配的 `GFP_KERNEL_ACCOUNT` 快捷键，其应该被核算。
- 用户空间的分配应该使用 `GFP_USER`、`GFP_HIGHUSER` 或 `GFP_HIGHUSER_MOVABLE` 中的一个标志。标志名称越长，限制性越小。

`GFP_HIGHUSER_MOVABLE` 不要求分配的内存将被内核直接访问，并意味着数据是可迁移的。

`GFP_HIGHUSER` 意味着所分配的内存是不可迁移的，但也不要求它能被内核直接访问。举个例子就是一个硬件分配内存，这些数据直接映射到用户空间，但没有寻址限制。

GFP\_USER 意味着分配的内存是不可迁移的，它必须被内核直接访问。

你可能会注意到，在现有的代码中，有相当多的分配指定了 GFP\_NOIO 或 GFP\_NOMS。从历史上看，它们被用来防止递归死锁，这种死锁是由直接内存回收调用到 FS 或 IO 路径以及对已经持有的资源进行阻塞引起的。从 4.12 开始，解决这个问题的首选方法是使用新的范围 API，即 Documentation/core-api/gfp\_mask-from-fs-io.rst.

其他传统的 GFP 标志是 GFP\_DMA 和 GFP\_DMA32。它们用于确保分配的内存可以被寻址能力有限的硬件访问。因此，除非你正在为一个有这种限制的设备编写驱动程序，否则要避免使用这些标志。而且，即使是有限制的硬件，也最好使用 dma\_alloc\* APIs。

### GFP 标志和回收行为

内存分配可能会触发直接或后台回收，了解页面分配器将如何努力满足该请求或其他请求是非常有用的。

- GFP\_KERNEL & \_\_GFP\_RECLAIM - 乐观分配，完全不尝试释放内存。最轻量级的模式，甚至不启动后台回收。应该小心使用，因为它可能会耗尽内存，而下一个用户可能会启动更积极的回收。
- GFP\_KERNEL & \_\_GFP\_DIRECT\_RECLAIM (or GFP\_NOWAIT) - 乐观分配，不试图从当前上下文中释放内存，但如果该区域低于低水位，可以唤醒 kswapd 来回收内存。可以从原子上下文中使用，或者当请求是一个性能优化，并且有另一个慢速路径的回退。
- (GFP\_KERNEL | \_\_GFP\_HIGH) & \_\_GFP\_DIRECT\_RECLAIM (aka GFP\_ATOMIC) - 非睡眠分配，有一个昂贵的回退，所以它可以访问某些部分的内存储备。通常从中断/底层上下文中使用，有一个昂贵的慢速路径回退。
- GFP\_KERNEL - 允许后台和直接回收，并使用默认的页面分配器行为。这意味着廉价的分配请求基本上是不会失败的，但不能保证这种行为，所以失败必须由调用者适当检查（例如，目前允许 OOM 杀手失败）。
- GFP\_KERNEL | \_\_GFP\_NORETRY - 覆盖默认的分配器行为，所有的分配请求都会提前失败，而不是导致破坏性的回收（在这个实现中是一轮的回收）。OOM 杀手不被调用。
- GFP\_KERNEL | \_\_GFP\_RETRY\_MAYFAIL - 覆盖默认的分配器行为，所有分配请求都非常努力。如果回收不能取得任何进展，该请求将失败。OOM 杀手不会被触发。
- GFP\_KERNEL | \_\_GFP\_NOFAIL - 覆盖默认的分配器行为，所有分配请求将无休止地循环，直到成功。这可能真的很危险，特别是对于较大的需求。

### 选择内存分配器

分配内存的最直接的方法是使用 kmalloc() 系列的函数。而且，为了安全起见，最好使用将内存设置为零的例程，如 kzalloc()。如果你需要为一个数组分配内存，有 kmalloc\_array() 和 kcalloc() 辅助程序。辅助程序 struct\_size()、array\_size() 和 array3\_size() 可以用来安全地计算对象的大小而不会溢出。

可以用 kmalloc 分配的块的最大尺寸是有限的。实际的限制取决于硬件和内核配置，但是对于小于页面大小的对象，使用 kmalloc 是一个好的做法。

用 *kmalloc* 分配的块的地址至少要对齐到 ARCH\_KMALLOC\_MINALIGN 字节。对于 2 的幂的大小，对齐方式也被保证为至少是各自的大小。

用 *kmalloc()* 分配的块可以用 *krealloc()* 调整大小。与 *kmalloc\_array()* 类似：以 *krealloc\_array()* 的形式提供了一个用于调整数组大小的辅助工具。

对于大量的分配，你可以使用 *vmalloc()* 和 *vzalloc()*，或者直接向页面分配器请求页面。由 *vmalloc* 和相关函数分配的内存在物理上是不连续的。

如果你不确定分配的大小对 *kmalloc* 来说是否太大，可以使用 *kvmalloc()* 及其派生函数。它将尝试用 *kmalloc* 分配内存，如果分配失败，将用 *vmalloc* 重新尝试。对于哪些 GFP 标志可以与 *kvmalloc* 一起使用是有限制的；请看 *kvmalloc\_node()* 参考文档。注意，*kvmalloc* 可能会返回物理上不连续的内存。

如果你需要分配许多相同的对象，你可以使用 slab 缓存分配器。在使用缓存之前，应该用 *kmem\_cache\_create()* 或 *kmem\_cache\_create\_usercopy()* 来设置缓存。如果缓存的一部分可能被复制到用户空间，应该使用第二个函数。在缓存被创建后，*kmem\_cache\_alloc()* 和它的封装可以从该缓存中分配内存。

当分配的内存不再需要时，它必须被释放。你可以使用 *kvfree()* 来处理用 *kmalloc*、*vmalloc* 和 *kvmalloc* 分配的内存。slab 缓存应该用 *kmem\_cache\_free()* 来释放。不要忘记用 *kmem\_cache\_destroy()* 来销毁缓存。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

**Original** Documentation/core-api/unaligned-memory-access.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 时奎亮 <[alexs@kernel.org](mailto:alexs@kernel.org)>

## 非对齐内存访问

作者 Daniel Drake <[dsd@gentoo.org](mailto:dsd@gentoo.org)>,

作者 Johannes Berg <[johannes@sipsolutions.net](mailto:johannes@sipsolutions.net)>

感谢他们的帮助 Alan Cox, Avuton Olrich, Heikki Orsila, Jan Engelhardt, Kyle McMarnin,  
Kyle Moffett, Randy Dunlap, Robert Hancock, Uli Kunitz, Vadim Lobanov

Linux 运行在各种各样的架构上，这些架构在内存访问方面有不同的表现。本文介绍了一些关于不对齐访问的细节，为什么你需要编写不引起不对齐访问的代码，以及如何编写这样的代码

### 非对齐访问的定义

当你试图从一个不被 N 偶数整除的地址（即  $\text{addr \% } N \neq 0$ ）开始读取 N 字节的数据时，就会发生无对齐内存访问。例如，从地址 0x10004 读取 4 个字节的数据是可以的，但从地址 0x10005 读取 4 个字节的数据将是一个不对齐的内存访问。

上述内容可能看起来有点模糊，因为内存访问可以以不同的方式发生。这里的背景是在机器码层面上：某些指令在内存中读取或写入一些字节（例如 x86 汇编中的 `movb`、`movw`、`movl`）。正如将变得清晰的那样，相对容易发现那些将编译为多字节内存访问指令的 C 语句，即在处理 `u16`、`u32` 和 `u64` 等类型时。

### 自然对齐

上面提到的规则构成了我们所说的自然对齐。当访问 N 个字节的内存时，基础内存地址必须被 N 平均分割，即  $\text{addr \% } N == 0$ 。

在编写代码时，假设目标架构有自然对齐的要求。

在现实中，只有少数架构在所有大小的内存访问上都要求自然对齐。然而，我们必须考虑所有支持的架构；编写满足自然对齐要求的代码是实现完全可移植性的最简单方法。

### 为什么非对齐访问时坏事

执行非对齐内存访问的效果因架构不同而不同。在这里写一整篇关于这些差异的文档是很容易的；下面是对常见情况的总结：

- 一些架构能够透明地执行非对齐内存访问，但通常会有很大的性能代价。
- 当不对齐的访问发生时，一些架构会引发处理器异常。异常处理程序能够纠正不对齐的访问，但要付出很大的性能代价。
- 一些架构在发生不对齐访问时，会引发处理器异常，但异常中并没有包含足够的信息来纠正不对齐访问。
- 有些架构不能进行无对齐内存访问，但会默默地执行与请求不同的内存访问，从而导致难以发现的微妙的代码错误！

从上文可以看出，如果你的代码导致不对齐的内存访问发生，那么你的代码在某些平台上将无法正常工作，在其他平台上将导致性能问题。

### 不会导致非对齐访问的代码

起初，上面的概念似乎有点难以与实际编码实践联系起来。毕竟，你对某些变量的内存地址没有很大的控制权，等等。

幸运的是事情并不复杂，因为在大多数情况下，编译器会确保代码工作正常。例如，以下面的结构体为例：

```
struct foo {
    u16 field1;
    u32 field2;
    u8 field3;
};
```

让我们假设上述结构体的一个实例驻留在从地址 0x10000 开始的内存中。根据基本的理解，访问 field2 会导致非对齐访问，这并不是不合理的。你会期望 field2 位于该结构体的 2 个字节的偏移量，即地址 0x10002，但该地址不能被 4 平均整除（注意，我们在这里读一个 4 字节的值）。

幸运的是，编译器理解对齐约束，所以在上述情况下，它会在 field1 和 field2 之间插入 2 个字节的填充。因此，对于标准的结构体类型，你总是可以依靠编译器来填充结构体，以便对字段的访问可以适当地对齐（假设你没有将字段定义不同长度的类型）。

同样，你也可以依靠编译器根据变量类型的大小，将变量和函数参数对齐到一个自然对齐的方案。

在这一点上，应该很清楚，访问单个字节（u8 或 char）永远不会导致无对齐访问，因为所有的内存地址都可以被 1 均匀地整除。

在一个相关的话题上，考虑到上述因素，你可以观察到，你可以对结构体中的字段进行重新排序，以便将字段放在不重排就会插入填充物的地方，从而减少结构体实例的整体常驻内存大小。上述例子的最佳布局是：

```
struct foo {
    u32 field2;
    u16 field1;
    u8 field3;
};
```

对于一个自然对齐方案，编译器只需要在结构的末尾添加一个字节的填充。添加这种填充是为了满足这些结构的数组的对齐约束。

另一点值得一提的是在结构体类型上使用 `_attribute_((packed))`。这个 GCC 特有的属性告诉编译器永远不要在结构体中插入任何填充，当你想用 C 结构体来表示一些“off the wire”的固定排列的数据时，这个属性很有用。

你可能会倾向于认为，在访问不满足架构对齐要求的字段时，使用这个属性很容易导致不对齐的访问。然而，编译器也意识到了对齐的限制，并且会产生额外的指令来执行内存访问，以避免造成不对齐的访问。当然，与 non-packed 的情况相比，额外的指令显然会造成性能上的损失，所以 packed 属性应该只在避免结构填充很重要的时候使用。

## 导致非对齐访问的代码

考虑到上述情况，让我们来看看一个现实生活中可能导致非对齐内存访问的函数的例子。下面这个函数取自 include/linux/etherdevice.h，是一个优化的例程，用于比较两个以太网 MAC 地址是否相等：

```
bool ether_addr_equal(const u8 *addr1, const u8 *addr2)
{
#define CONFIG_HAVE_EFFICIENT_UNALIGNED_ACCESS
    u32 fold = ((*(const u32 *)addr1) ^ (*(const u32 *)addr2)) |
               ((*(const u16 *)(addr1 + 4)) ^ (*(const u16 *)(addr2 + 4)));

    return fold == 0;
#else
    const u16 *a = (const u16 *)addr1;
    const u16 *b = (const u16 *)addr2;
    return ((a[0] ^ b[0]) | (a[1] ^ b[1]) | (a[2] ^ b[2])) == 0;
#endif
}
```

在上述函数中，当硬件具有高效的非对齐访问能力时，这段代码没有问题。但是当硬件不能在任意边界上访问内存时，对 a[0] 的引用导致从地址 addr1 开始的 2 个字节（16 位）被读取。

想一想，如果 addr1 是一个奇怪的地址，如 0x10003，会发生什么？（提示：这将是一个非对齐访问。）

尽管上述函数存在潜在的非对齐访问问题，但它还是被包含在内核中，但被理解为只在 16 位对齐的地址上正常工作。调用者应该确保这种对齐方式或者根本不使用这个函数。这个不对齐的函数仍然是有用的，因为它是在你能确保对齐的情况下一个很好的优化，这在以太网网络环境中几乎是一直如此。

下面是另一个可能导致非对齐访问的代码的例子：

```
void myfunc(u8 *data, u32 value)
{
    [...]
    *((u32 *) data) = cpu_to_le32(value);
    [...]
}
```

每当数据参数指向的地址不被 4 均匀整除时，这段代码就会导致非对齐访问。

综上所述，你可能遇到非对齐访问问题的两种主要情况包括：

1. 将变量定义不同长度的类型
2. 指针运算后访问至少 2 个字节的数据

## 避免非对齐访问

避免非对齐访问的最简单方法是使用 `<asm/unaligned.h>` 头文件提供的 `get_unaligned()` 和 `put_unaligned()` 宏。

回到前面的一个可能导致非对齐访问的代码例子：

```
void myfunc(u8 *data, u32 value)
{
    [...]
    *((u32 *) data) = cpu_to_le32(value);
    [...]
}
```

为了避免非对齐的内存访问，你可以将其改写如下：

```
void myfunc(u8 *data, u32 value)
{
    [...]
    value = cpu_to_le32(value);
    put_unaligned(value, (u32 *) data);
    [...]
}
```

`get_unaligned()` 宏的工作原理与此类似。假设' `data`' 是一个指向内存的指针，并且你希望避免非对齐访问，其用法如下：

```
u32 value = get_unaligned((u32 *) data);
```

这些宏适用于任何长度的内存访问（不仅仅是上面例子中的 32 位）。请注意，与标准的对齐内存访问相比，使用这些宏来访问非对齐内存可能会在性能上付出代价。

如果使用这些宏不方便，另一个选择是使用 `memcpy()`，其中源或目标（或两者）的类型为 `u8*` 或非对齐 `char*`。由于这种操作的字节性质，避免了非对齐访问。

## 对齐 vs. 网络

在需要对齐负载的架构上，网络要求 IP 头在四字节边界上对齐，以优化 IP 栈。对于普通的以太网硬件，常数 `NET_IP_ALIGN` 被使用。在大多数架构上，这个常数的值是 2，因为正常的以太网头是 14 个字节，所以为了获得适当的对齐，需要 DMA 到一个可以表示为  $4*n+2$  的地址。一个值得注意的例外是 `powerpc`，它将 `NET_IP_ALIGN` 定义为 0，因为 DMA 到未对齐的地址可能非常昂贵，与未对齐的负载的成本相比相形见绌。

对于一些不能 DMA 到未对齐地址的以太网硬件，如  $4*n+2$  或非以太网硬件，这可能是一个问题，这时需要将传入的帧复制到一个对齐的缓冲区。因为这在可以进行非对齐访问的架构上是不必要的，所以可以使代码依赖于 `CONFIG_HAVE_EFFICIENT_UNALIGNED_ACCESS`，像这样：

```
#ifdef CONFIG_HAVE_EFFICIENT_UNALIGNED_ACCESS
    skb = original_skb
#else
    skb = copy_skb
#endif
```

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

### Original Documentation/core-api/mm-api.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 时奎亮 <[alexs@kernel.org](mailto:alexs@kernel.org)>

## 内存管理 APIs

API (Application Programming Interface, 应用程序接口)

### 用户空间内存访问

该 API 在以下内核代码中：

arch/x86/include/asm/uaccess.h

arch/x86/lib/usercopy\_32.c

mm/gup.c

### 内存分配控制

该 API 在以下内核代码中：

include/linux/gfp.h

## Slab 缓存

此缓存非 cpu 片上缓存， 请读者自行查阅资料。

该 API 在以下内核代码中:

include/linux/slab.h

mm/slab.c

mm/slab\_common.c

mm/util.c

## 虚拟连续（内存页）映射

该 API 在以下内核代码中:

mm/vmalloc.c

## 文件映射和页面缓存

该 API 在以下内核代码中:

mm/readahead.c

mm/filemap.c

mm/page-writeback.c

mm/truncate.c

include/linux/pagemap.h

## 内存池

该 API 在以下内核代码中:

mm/mempool.c

## DMA 池

DMA(Direct Memory Access, 直接存储器访问)

该 API 在以下内核代码中:

mm/dmapool.c

### 更多的内存管理函数

该 API 在以下内核代码中:

mm/memory.c

mm/page\_alloc.c

mm/mempolicy.c

include/linux/mm\_types.h

include/linux/mm.h

include/linux/mmzone.h

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/core-api/genalloc.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 时奎亮 <[alexs@kernel.org](mailto:alexs@kernel.org)>

### genalloc/genpool 子系统

内核中有许多内存分配子系统，每一个都是针对特定的需求。然而，有时候，内核开发者需要为特定范围的特殊用途的内存实现一个新的分配器；通常这个内存位于某个设备上。该设备的驱动程序的作者当然可以写一个小的分配器来完成工作，但这是让内核充满几十个测试差劲的分配器的方法。早在 2005 年，Jes Sorensen 从 sym53c8xx\_2 驱动中提取了其中的一个分配器，并将其作为一个通用模块发布，用于创建特设的内存分配器。这段代码在 2.6.13 版本中被合并；此后它被大大地修改了。

使用这个分配器的代码应该包括 <linux/genalloc.h>。这个动作从创建一个池开始，使用一个：

该 API 在以下内核代码中:

lib/genalloc.c

对 gen\_pool\_create() 的调用将创建一个内存池。分配的粒度由 min\_alloc\_order 设置；它是一个 log-base-2（以 2 为底的对数）的数字，就像页面分配器使用的数字一样，但它指的是字节而不是页面。因此，如果 min\_alloc\_order 被传递为 3，那么所有的分配将是 8 字节的倍数。增加 min\_alloc\_order 可以减少跟

踪池中内存所需的内存。`nid` 参数指定哪一个 NUMA 节点应该被用于分配管家结构体；如果调用者不关心，它可以是-1。

“管理的”接口 `devm_gen_pool_create()` 将内存池与一个特定的设备联系起来。在其他方面，当给定的设备被销毁时，它将自动清理内存池。

一个内存池池被关闭的方法是：

该 API 在以下内核代码中：

`lib/genalloc.c`

值得注意的是，如果在给定的内存池中仍有未完成的分配，这个函数将采取相当极端的步骤，调用 `BUG()`，使整个系统崩溃。你已经被警告了。

一个新创建的内存池没有内存可以分配。在这种状态下，它是相当无用的，所以首要任务之一通常是向内存池里添加内存。这可以通过以下方式完成：

该 API 在以下内核代码中：

`include/linux/genalloc.h`

`lib/genalloc.c`

对 `gen_pool_add()` 的调用将把从地址（在内核的虚拟地址空间）开始的内存的大小字节放入给定的池中，再次使用 `nid` 作为节点 ID 进行辅助内存分配。`gen_pool_add_virt()` 变体将显式物理地址与内存联系起来；只有在内存池被用于 DMA 分配时，这才是必要的。

从内存池中分配内存（并将其放回）的函数是：

该 API 在以下内核代码中：

`include/linux/genalloc.h`

`lib/genalloc.c`

正如人们所期望的，`gen_pool_alloc()` 将从给定的池中分配 `size` 字节。`gen_pool_dma_alloc()` 变量分配内存用于 DMA 操作，返回 `dma` 所指向的空间中的相关物理地址。这只有在内存是用 `gen_pool_add_virt()` 添加的情况下才会起作用。请注意，这个函数偏离了 `genpool` 通常使用无符号长值来表示内核地址的模式；它返回一个 `void *` 来代替。

这一切看起来都比较简单；事实上，一些开发者显然认为这太简单了。毕竟，上面的接口没有提供对分配函数如何选择返回哪块特定内存的控制。如果需要这样的控制，下面的函数将是有意义的：

该 API 在以下内核代码中：

`lib/genalloc.c`

使用 `gen_pool_alloc_algo()` 进行的分配指定了一种用于选择要分配的内存的算法；默认算法可以用 `gen_pool_set_algo()` 来设置。数据值被传递给算法；大多数算法会忽略它，但偶尔也会需要它。当然，人们可以写一个特殊用途的算法，但是已经有一套公平的算法可用：

- `gen_pool_first_fit` 是一个简单的初配分配器；如果没有指定其他算法，这是默认算法。

- `gen_pool_first_fit_align` 强迫分配有一个特定的对齐方式（通过 `genpool_data_align` 结构中的数据传递）。
- `gen_pool_first_fit_order_align` 按照大小的顺序排列分配。例如，一个 60 字节的分配将以 64 字节对齐。
- `gen_pool_best_fit`, 正如人们所期望的，是一个简单的最佳匹配分配器。
- `gen_pool_fixed_alloc` 在池中的一个特定偏移量（通过数据参数在 `genpool_data_fixed` 结构中传递）进行分配。如果指定的内存不可用，则分配失败。

还有一些其他的函数，主要是为了查询内存池中的可用空间或迭代内存块等目的。然而，大多数用户应该不需要以上描述的功能。如果幸运的话，对这个模块的广泛认识将有助于防止在未来编写特殊用途的内存分配器。

该 API 在以下内核代码中：

`lib/genalloc.c`

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:<alexs@kernel.org>)。

---

**Original** Documentation/core-api/boot-time-mm.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 时奎亮 <[alexs@kernel.org](mailto:alexs@kernel.org)>

### 启动时的内存管理

系统初始化早期“正常”的内存管理由于没有设置完毕无法使用。但是内核仍然需要为各种数据结构分配内存，例如物理页分配器。

一个叫做 `memblock` 的专用分配器执行启动时的内存管理。特定架构的初始化必须在 `setup_arch()` 中设置它，并在 `mem_init()` 函数中移除它。

一旦早期的内存管理可用，它就为内存分配提供了各种函数和宏。分配请求可以指向第一个（也可能是唯一的）节点或 NUMA 系统中的某个特定节点。有一些 API 变体在分配失败时 `panic`，也有一些不会 `panic` 的。

`Memblock` 还提供了各种控制其自身行为的 API。

## Memblock 概述

该 API 在以下内核代码中:

mm/memblock.c

## 函数和结构体

下面是关于 membblock 数据结构、函数和宏的描述。其中一些实际上是内部的，但由于它们被记录下来，漏掉它们是很愚蠢的。此外，阅读内部函数的注释可以帮助理解引擎盖下真正发生的事情。

该 API 在以下内核代码中:

include/linux/memblock.h mm/memblock.c

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/core-api/gfp\_mask-from-fs-io.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 时奎亮 <[alexs@kernel.org](mailto:alexs@kernel.org)>

## 从 FS/IO 上下文中使用的 GFP 掩码

日期 2018 年 5 月

作者 Michal Hocko <[mhocko@kernel.org](mailto:mhocko@kernel.org)>

## 简介

文件系统和 IO 栈中的代码路径在分配内存时必须小心，以防止因直接调用 FS 或 IO 路径的内存回收和阻塞已经持有的资源（例如锁-最常见的是用于事务上下文的锁）而造成递归死锁。

避免这种死锁问题的传统方法是在调用分配器时，在 gfp 掩码中清除 \_\_GFP\_FS 和 \_\_GFP\_IO（注意后者意味着也要清除第一个）。GFP\_NOFS 和 GFP\_NOIO 可以作为快捷方式使用。但事实证明，上述方法导致了滥用，当限制性的 gfp 掩码被用于“万一”时，没有更深入的考虑，这导致了问题，因为过度使用 GFP\_NOFS/GFP\_NOIO 会导致内存过度回收或其他内存回收的问题。

### 新 API

从 4.12 开始，我们为 NOFS 和 NOIO 上下文提供了一个通用的作用域 API，分别是 `memalloc_nofs_save`, `memalloc_nofs_restore` 和 `memalloc_noio_save`, `memalloc_noio_restore`，允许从文件系统或 I/O 的角度将一个作用域标记为一个关键部分。从该作用域的任何分配都将从给定的掩码中删除 `_GFP_FS` 和 `_GFP_IO`，所以没有内存分配可以追溯到 FS/IO 中。

该 API 在以下内核代码中：

```
include/linux/sched/mm.h
```

然后，FS/IO 代码在任何与回收有关的关键部分开始之前简单地调用适当的保存函数——例如，与回收上下文共享的锁或当事务上下文嵌套可能通过回收进行时。恢复函数应该在关键部分结束时被调用。所有这一切最好都伴随着解释什么是回收上下文，以方便维护。

请注意，保存/恢复函数的正确配对允许嵌套，所以从现有的 NOIO 或 NOFS 范围分别调用 `memalloc_noio_save` 或 `memalloc_noio_restore` 是安全的。

### 那么 `_vmalloc(GFP_NOFS)` 呢？

`vmalloc` 不支持 `GFP_NOFS` 语义，因为在分配器的深处有硬编码的 `GFP_KERNEL` 分配，要修复这些分配是相当不容易的。这意味着用 `GFP_NOFS/GFP_NOIO` 调用 `vmalloc` 几乎总是一个错误。好消息是，`NOFS/NOIO` 语义可以通过范围 API 实现。

在理想的世界中，上层应该已经标记了危险的上下文，因此不需要特别的照顾，`vmalloc` 的调用应该没有任何问题。有时，如果上下文不是很清楚，或者有叠加的违规行为，那么推荐的方法是用范围 API 包装 `vmalloc`，并加上注释来解释问题。

Todolist:

```
dma-api dma-api-howto dma-attributes dma-isa-lpc pin_user_pages
```

### 内核调试的接口

Todolist:

```
debug-objects tracepoint debugging-via-ohci1394
```

### 其它文档

不适合放在其它地方或尚未归类的文件；

Todolist:

```
librs
```

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

## Original Documentation/locking/index.rst

翻译 唐艺舟 Tang Yizhou <[tangyeechou@gmail.com](mailto:tangyeechou@gmail.com)>

### \* 锁

TODOList:

- locktypes
- lockdep-design
- lockstat
- locktorture
- mutex-design
- rt-mutex-design
- rt-mutex
- seqlock
- spinlocks
- ww-mutex-design
- preempt-locking
- pi-futex
- futex-requeue-pi
- hwspinlock
- percpu-rw-semaphore
- robust-futexes
- robust-futex-ABI

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/accounting/index.rst

**Translator** Yang Yang <[yang.yang29@zte.com.cn](mailto:yang.yang29@zte.com.cn)>

### \* 计数

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/accounting/delay-accounting.rst

**Translator** Yang Yang <[yang.yang29@zte.com.cn](mailto:yang.yang29@zte.com.cn)>

### 延时计数

任务在等待某些内核资源可用时，会造成延时。例如一个可运行的任务可能会等待一个空闲 CPU 来运行。

基于每任务的延时计数功能度量由以下情况造成任务延时：

- a) 等待一个 CPU (任务为可运行)
- b) 完成由该任务发起的块 I/O 同步请求
- c) 页面交换
- d) 内存回收
- e) 页缓存抖动
- f) 直接规整

并将这些统计信息通过 taskstats 接口提供给用户空间。

这些延时信息为适当的调整任务 CPU 优先级、io 优先级、rss 限制提供反馈。重要任务长期延时，表示可能需要提高其相关优先级。

通过使用 taskstats 接口，本功能还可提供一个线程组（对应传统 Unix 进程）所有任务（或线程）的总延时统计信息。此类汇总往往是需要的，由内核来完成更加高效。

用户空间的实体，特别是资源管理程序，可将延时统计信息汇总到任意组中。为实现这一点，任务的延时统计信息在其生命周期内和退出时皆可获取，从而确保可进行连续、完整的监控。

## 接口

延时计数使用 taskstats 接口，该接口由本目录另一个单独的文档详细描述。Taskstats 向用户态返回一个通用数据结构，对应每 pid 或每 tgid 的统计信息。延时计数功能填写该数据结构的特定字段。见

```
include/uapi/linux/taskstats.h
```

其描述了延时计数相关字段。系统通常以计数器形式返回 CPU、同步块 I/O、交换、内存回收、页缓存抖动、直接规整等的累积延时。

取任务某计数器两个连续读数的差值，将得到任务在该时间间隔内等待对应资源的总延时。

当任务退出时，内核会将包含每任务的统计信息发送给用户空间，而无需额外的命令。若其为线程组最后一个退出的任务，内核还会发送每 tgid 的统计信息。更多详细信息见 taskstats 接口的描述。

tools/accounting 目录中的用户空间程序 getdelays.c 提供了一些简单的命令，用以显示延时统计信息。其也是使用 taskstats 接口的示例。

## 用法

使用以下配置编译内核：

```
CONFIG_TASK_DELAY_ACCT=y
CONFIG_TASKSTATS=y
```

延时计数在启动时默认关闭。若需开启，在启动参数中增加：

```
delayacct
```

本文后续的说明基于延时计数已开启。也可在系统运行时，使用 sysctl 的 kernel.task\_delayacct 进行开关。注意，只有在启用延时计数后启动的任务才会有相关信息。

系统启动后，使用类似 getdelays.c 的工具获取任务或线程组（tgid）的延时信息。

getdelays 命令的一般格式：

```
getdelays [-dilv] [-t tgid] [-p pid]
```

获取 pid 为 10 的任务从系统启动后的延时信息：

```
# ./getdelays -d -p 10  
(输出信息和下例相似)
```

获取所有 tgid 为 5 的任务从系统启动后的总延时信息:

```
# ./getdelays -d -t 5  
print delayacct stats ON  
Tgid      5  
  
CPU          count    real total   virtual total   delay total   delay average  
          8           7000000       6872122       3382277       0.423ms  
IO          count    delay total   delay average  
          0           0             0ms  
SWAP         count    delay total   delay average  
          0           0             0ms  
RECLAIM      count    delay total   delay average  
          0           0             0ms  
THRASHING     count    delay total   delay average  
          0           0             0ms  
COMPACT       count    delay total   delay average  
          0           0             0ms
```

获取 **pid** 为 1 的 **IO** 计数, 它只和**-p** 一起使用:: # ./getdelays -i -p 1 printing IO accounting linuxrc:  
read=65536, write=0, cancelled\_write=0

上面的命令与**-v** 一起使用, 可以获取更多调试信息。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解, 而不是作为一个分支。因此, 如果您对此文件有任何意见或更新, 请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题, 请联系该文件的译者, 或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/accounting/psi.rst

**Translator** Yang Yang <[yang.yang29@zte.com.cn](mailto:yang.yang29@zte.com.cn)>

## PSI——压力阻塞信息

日期 April, 2018

作者 Johannes Weiner <[hannes@cmpxchg.org](mailto:hannes@cmpxchg.org)>

当 CPU、memory 或 IO 设备处于竞争状态，业务负载会遭受时延毛刺、吞吐量降低，及面临 OOM 的风险。如果没有一种准确的方法度量系统竞争程度，则有两种后果：一种是用户过于节制，未充分利用系统资源；另一种是过度使用，经常性面临业务中断的风险。

psi 特性能够识别和量化资源竞争导致的业务中断，及其对复杂负载乃至整个系统在时间上的影响。

准确度量因资源不足造成的生产力损失，有助于用户基于硬件调整业务负载，或基于业务负载配置硬件。

psi 能够实时的提供相关信息，因此系统可基于 psi 实现动态的负载管理。如实施卸载、迁移、策略性的停止或杀死低优先级或可重启的批处理任务。

psi 帮助用户实现硬件资源利用率的最大化。同时无需牺牲业务负载健康度，也无需面临 OOM 等造成业务中断的风险。

## 压力接口

压力信息可通过/proc/pressure/ -cpu、memory、io 文件分别获取。

CPU 相关信息格式如下：

```
some avg10=0.00 avg60=0.00 avg300=0.00 total=0
```

内存和 IO 相关信息如下：

```
some avg10=0.00 avg60=0.00 avg300=0.00 total=0 full avg10=0.00 avg60=0.00
avg300=0.00 total=0
```

some 行代表至少有一个任务阻塞于特定资源的时间占比。

full 行代表所有非 idle 任务同时阻塞于特定资源的时间占比。在这种状态下 CPU 资源完全被浪费，相对于正常运行，业务负载由于耗费更多时间等待而受到严重影响。

由于此情况严重影响系统性能，因此清楚的识别本情况并与 some 行所代表的情况区分开，将有助于分析及提升系统性能。这就是 full 独立于 some 行的原因。

avg 代表阻塞时间占比（百分比），为最近 10 秒、60 秒、300 秒内的均值。这样我们既可观察到短期事件的影响，也可看到中等及长时间内的趋势。total 代表总阻塞时间（单位微秒），可用于观察时延毛刺，这种毛刺可能在均值中无法体现。

## 监控压力门限

用户可注册触发器，通过 poll() 监控资源压力是否超过门限。

触发器定义：指定时间窗口期内累积阻塞时间的最大值。比如可定义 500ms 内积累 100ms 阻塞，即触发一次唤醒事件。

触发器注册方法：用户打开代表特定资源的 psi 接口文件，写入门限、时间窗口的值。所打开的文件描述符用于等待事件，可使用 select()、poll()、epoll()。写入信息的格式如下：

```
<some|full> <stall amount in us> <time window in us>
```

示例：向/proc/pressure/memory 写入 “some 150000 1000000” 将新增触发器，将在 1 秒内至少一个任务阻塞于内存的总时间超过 150ms 时触发。向/proc/pressure/io 写入 “full 50000 1000000” 将新增触发器，将在 1 秒内所有任务都阻塞于 io 的总时间超过 50ms 时触发。

触发器可针对多个 psi 度量值设置，同一个 psi 度量值可设置多个触发器。每个触发器需要单独的文件描述符用于轮询，以区别于其他触发器。所以即使对于同一个 psi 接口文件，每个触发器也需要单独的调用 open()。

监控器在被监控资源进入阻塞状态时启动，在系统退出阻塞状态后停用。系统进入阻塞状态后，监控 psi 增长的频率为每监控窗口刷新 10 次。

内核接受的窗口为 500ms~10s，所以监控间隔为 50ms~1s。设置窗口下限目的是为了防止过于频繁的轮询。设置窗口上限的目的是因为窗口过长则无意义，此时查看 psi 接口提供的均值即可。

监控器在激活后，至少在跟踪窗口期间将保持活动状态。以避免随着系统进入和退出阻塞状态，监控器过于频繁的进入和退出活动状态。

用户态通知在监控窗口内会受到速率限制。当对应的文件描述符关闭，触发器会自动注销。

## 用户态监控器使用示例

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <poll.h>
#include <string.h>
#include <unistd.h>

/* 监控内存部分阻塞，监控时间窗口为 1 秒、阻塞门限为 150 毫秒。 */
int main() {
    const char trig[] = "some 150000 1000000";
    struct pollfd fds;
    int n;

    fds.fd = open("/proc/pressure/memory", O_RDWR | O_NONBLOCK);
    if (fds.fd < 0) {
        printf("/proc/pressure/memory open error: %s\n",
               strerror(errno));
        return 1;
    }

    /* ... (polling loop) ... */

    close(fds.fd);
}
```

```

}

fds.events = POLLPRI;

if (write(fds.fd, trig, strlen(trig) + 1) < 0) {
    printf("/proc/pressure/memory write error: %s\n",
           strerror(errno));
    return 1;
}

printf("waiting for events...\n");
while (1) {
    n = poll(&fds, 1, -1);
    if (n < 0) {
        printf("poll error: %s\n", strerror(errno));
        return 1;
    }
    if (fds.revents & POLLERR) {
        printf("got POLLERR, event source is gone\n");
        return 0;
    }
    if (fds.revents & POLLPRI) {
        printf("event triggered!\n");
    } else {
        printf("unknown event received: 0x%llx\n", fds.revents);
        return 1;
    }
}
return 0;
}

```

## Cgroup2 接口

对于 CONFIG\_CGROUP=y 及挂载了 cgroup2 文件系统的系统，能够获取 cgroups 内任务的 psi。此场景下 cgroupfs 挂载点的子目录包含 cpu.pressure、memory.pressure、io.pressure 文件，内容格式与/proc/pressure/下的文件相同。

可设置基于 cgroup 的 psi 监控器，方法与系统级 psi 监控器相同。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/accounting/taskstats.rst

**Translator** Yang Yang <[yang.yang29@zte.com.cn](mailto:yang.yang29@zte.com.cn)>

### 每任务的统计接口

Taskstats 是一个基于 netlink 的接口，用于从内核向用户空间发送每任务及每进程的统计信息。

Taskstats 设计目的：

- 在任务生命周期内和退出时高效的提供统计信息
- 统一不同计数子系统的接口
- 支持未来计数系统的扩展

### 术语

“pid”、“tid”、“任务”互换使用，用于描述由 struct task\_struct 定义的标准 Linux 任务。“每 pid 的统计数据”等价于“每任务的统计数据”。

“tgid”、“进程”、“线程组”互换使用，用于描述共享 mm\_struct 的任务集，也就是传统的 Unix 进程。尽管使用了 tgid 这个词，即使一个任务是线程组组长，对它的处理也没有什么不同。只要一个进程还有任何归属它的任务，它就被认为活着。

### 用法

为了在任务生命周期内获得统计信息，用户空间需打开一个单播的 netlink 套接字（NETLINK\_GENERIC 族）然后发送指定 pid 或 tgid 的命令。响应消息中包含单个任务的统计信息（若指定了 pid）或进程所有任务汇总的统计信息（若指定了 tgid）。

为了在任务退出时获取统计信息，用户空间的监听者发送一个指定 cpu 掩码的注册命令。cpu 掩码内的 cpu 上有任务退出时，每 pid 的统计信息将发送给注册成功的监听者。使用 cpu 掩码可以限制一个监听者收到的数据，并有助于对 netlink 接口进行流量控制，后文将进行更详细的解释。

如果正在退出的任务是线程组中最后一个退出的线程，额外一条包含每 tgid 统计信息的记录也将发送给用户空间。后者包含线程组中所有线程（包括过去和现在）的每 pid 统计信息总和。

getdelays.c 是一个简单的示例，用以演示如何使用 taskstats 接口获取延迟统计信息。用户可注册 cpu 掩码、发送命令和处理响应、监听每 tid/tgid 退出数据、将收到的数据写入文件、通过增大接收缓冲区进行基本的流量控制。

## 接口

内核用户接口封装在 include/linux/taskstats.h。

为避免本文档随着接口的演进而过期，本文仅给出当前版本的概要。当本文与 taskstats.h 不一致时，以 taskstats.h 为准。

struct taskstats 是每 pid 和每 tgid 数据共用的计数结构体。它是版本化的，可在内核新增计数子系统时进行扩展。taskstats.h 中定义了各字段及语义。

用户、内核空间的数据交换是属于 NETLINK\_GENERIC 族的 netlink 消息，使用 netlink 属性接口。消息格式如下：

+-----+ - - +-----+-----+
nlmsghdr   Pad   genlmsghdr   taskstats payload
+-----+ - - +-----+-----+

Taskstats 载荷有三种类型：

1. 命令：由用户发送给内核。获取指定 pid/tgid 数据的命令包含一个类型为 TASKSTATS\_CMD\_ATTR\_PID/TGID 的属性，该属性包含 u32 的 pid 或 tgid 载荷。pid/tgid 指示用户空间要统计的任务/进程。

注册/注销 获取 指定 cpu 集上退出数据的命令包含一个类型为 TASKSTATS\_CMD\_ATTR\_REGISTER/DEREGISTER\_CPUMASK 的属性，该属性包含 cpu 掩码载荷。cpu 掩码是以 ascii 码表示，用逗号分隔的 cpu 范围。例如若需监听 1,2,3,5,7,8 号 cpu 的退出数据，cpu 掩码表示为“1-3,5,7-8”。若用户空间在关闭监听套接字前忘了注销监听的 cpu 集，随着时间的推移，内核会清理此监听集。但是，出于提效的目的，建议明确执行注销。

2. 命令的应答：内核发出应答用户空间的命令。载荷有三类属性：

- a) TASKSTATS\_TYPE\_AGGR\_PID/TGID：本属性不包含载荷，用以指示其后为被统计对象的 pid/tgid。
- b) TASKSTATS\_TYPE\_PID/TGID：本属性的载荷为 pid/tgid，其统计信息将被返回。
- c) TASKSTATS\_TYPE\_STATS：本属性的载荷为一个 struct taskstats 实例。每 pid 和每 tgid 统计信息共用该结构体。

3. 内核会在任务退出时发送新消息。其载荷包含一系列以下类型的属性：

- a) TASKSTATS\_TYPE\_AGGR\_PID：指示其后两个属性为 pid+stats。
- b) TASKSTATS\_TYPE\_PID：包含退出任务的 pid。
- c) TASKSTATS\_TYPE\_STATS：包含退出任务的每 pid 统计信息
- d) TASKSTATS\_TYPE\_AGGR\_TGID：指示其后两个属性为 tgid+stats。
- e) TASKSTATS\_TYPE\_TGID：包含任务所属进程的 tgid
- f) TASKSTATS\_TYPE\_STATS：包含退出任务所属进程的每 tgid 统计信息

### 每 tgid 的统计

除了每任务的统计信息，taskstats 还提供每进程的统计信息，因为资源管理通常以进程粒度完成，并且仅在用户空间聚合任务统计信息效率低下且可能不准确（缺乏原子性）。

然而，除了每任务统计信息，在内核中维护每进程统计信息存在额外的时间和空间开销。为解决此问题，taskstats 代码将退出任务的统计信息累积到进程范围的数据结构中。当进程最后一个任务退出时，累积的进程级数据也会发送到用户空间（与每任务数据一起）。

当用户查询每 tgid 数据时，内核将指定线程组中所有活动线程的统计信息相加，并添加到该线程组的累积总数（含之前退出的线程）。

### 扩展 taskstats

有两种方法可在未来修改内核扩展 taskstats 接口，以导出更多的每任务/进程统计信息：

1. 在现有 struct taskstats 末尾增加字段。该结构体中的版本号确保了向后兼容性。用户空间将仅使用与其版本对应的结构体字段。
2. 定义单独的统计结构体并使用 netlink 属性接口返回对应的数据。由于用户空间独立处理每个 netlink 属性，所以总是可以忽略其不理解类型的属性（因为使用了旧版本接口）。

在 1. 和 2. 之间进行选择，属于权衡灵活性和开销的问题。若仅需增加少数字段，那么 1. 是首选方法，因为内核和用户空间无需承担处理新 netlink 属性的开销。但若新字段过多的扩展现有结构体，导致不同的用户空间计数程序不必要的接收大型结构体，而对结构体字段并不感兴趣，那么 2. 是值得的。

### Taskstats 的流量控制

当退出任务数速率变大，监听者可能跟不上内核发送每 tid/tgid 退出数据的速率，而导致数据丢失。taskstats 结构体变大、cpu 数量上升，都会导致这种可能性增加。

为避免统计信息丢失，用户空间应执行以下操作中至少一项：

- 增大监听者用于接收退出数据的 netlink 套接字接收缓存区。
- 创建更多的监听者，减少每个监听者监听的 cpu 数量。极端情况下可为每个 cpu 创建一个监听者。用户还可考虑将监听者的 cpu 亲和性设置为监听 cpu 的子集，特别是当他们仅监听一个 cpu。

尽管采取了这些措施，若用户空间仍收到指示接收缓存区溢出的 ENOBUFS 错误消息，则应采取其他措施处理数据丢失。

Todolist:

cgroupstats taskstats-struct

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题, 请联系该文件的译者, 或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/cpu-freq/index.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## \* Linux CPUFreq - Linux(TM) 内核中的 CPU 频率和电压升降代码

Author: Dominik Brodowski <[linux@brodo.de](mailto:linux@brodo.de)>

时钟升降允许你在运行中改变 CPU 的时钟速度。这是一个很好的节省电池电量的方法, 因为时钟速度越低, CPU 消耗的电量越少。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解, 而不是作为一个分支。因此, 如果您对此文件有任何意见或更新, 请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题, 请联系该文件的译者, 或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/cpu-freq/core.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 唐艺舟 Tang Yizhou <[tangyeechou@gmail.com](mailto:tangyeechou@gmail.com)>

## CPUFreq 核心和 CPUFreq 通知器的通用说明

作者:

- Dominik Brodowski <[linux@brodo.de](mailto:linux@brodo.de)>
- David Kimdon <[dwhedon@debian.org](mailto:dwhedon@debian.org)>
- Rafael J. Wysocki <[rafael.j.wysocki@intel.com](mailto:rafael.j.wysocki@intel.com)>
- Viresh Kumar <[viresh.kumar@linaro.org](mailto:viresh.kumar@linaro.org)>

## 1. CPUPerf 核心和接口

cpufreq 核心代码位于 drivers/cpufreq/cpufreq.c 中。这些 cpufreq 代码为 CPUPerf 架构的驱动程序（那些执行硬件频率切换的代码）以及“通知器”提供了一个标准化的接口。包括设备驱动程序；需要了解策略变化（如 ACPI 热量管理），或所有频率变化（如计时代码），甚至需要强制限制为指定频率（如 ARM 架构上的 LCD 驱动程序）的其它内核组件。此外，内核“常数”loops\_per\_jiffy 会根据频率变化而更新。

cpufreq 策略的引用计数由 cpufreq\_cpu\_get 和 cpufreq\_cpu\_put 来完成，以确保 cpufreq 驱动程序被正确地注册到核心中，并且驱动程序在 cpufreq\_put\_cpu 被调用之前不会被卸载。这也保证了每个 CPU 核的 cpufreq 策略在使用期间不会被释放。

## 2. CPUPerf 通知器

CPUPerf 通知器遵循标准的内核通知器接口。关于通知器的细节请参阅 linux/include/linux/notifier.h。

这里有两个不同的 CPUPerf 通知器 - 策略通知器和转换通知器。

### 2.1 CPUPerf 策略通知器

当创建或移除策略时，这些都会被通知。

阶段是在通知器的第二个参数中指定的。当第一次创建策略时，阶段是 CPUFREQ\_CREATE\_POLICY，当策略被移除时，阶段是 CPUFREQ\_REMOVE\_POLICY。

第三个参数 void \*pointer 指向一个结构体 cpufreq\_policy，其包括 min, max(新策略的下限和上限（单位为 kHz）) 这几个值。

### 2.2 CPUPerf 转换通知器

当 CPUPerf 驱动切换 CPU 核心频率时，策略中的每个在线 CPU 都会收到两次通知，这些变化没有任何外部干预。

第二个参数指定阶段 - CPUFREQ\_PRECHANGE or CPUFREQ\_POSTCHANGE.

第三个参数是一个包含如下值的结构体 cpufreq\_freqs:

policy	指向 struct cpufreq_policy 的指针
old	旧频率
new	新频率
flags	cpufreq 驱动的标志

### 3. 含有 Operating Performance Point (OPP) 的 CPUFreq 表的生成

关于 OPP 的细节请参阅 Documentation/power/opp.rst

**dev\_pm\_opp\_init\_cpufreq\_table** - 这个函数提供了一个随时可用的转换例程，用来将 OPP 层关于可用频率的内部信息翻译成一种 cpufreq 易于处理的格式。

**Warning:** 不要在中断上下文中使用此函数。

例如：

```
soc_pm_init()
{
    /* Do things */
    r = dev_pm_opp_init_cpufreq_table(dev, &freq_table);
    if (!r)
        policy->freq_table = freq_table;
    /* Do other things */
}
```

**Note:** 该函数只有在 CONFIG\_PM\_OPP 之外还启用了 CONFIG\_CPU\_FREQ 时才可用。

**dev\_pm\_opp\_free\_cpufreq\_table** 释放 dev\_pm\_opp\_init\_cpufreq\_table 分配的表。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/cpu-freq/cpu-drivers.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 唐艺舟 Tang Yizhou <[tangyeechou@gmail.com](mailto:tangyeechou@gmail.com)>

### 如何实现一个新的 CPUFreq 处理器驱动程序?

作者:

- Dominik Brodowski <[linux@brodo.de](mailto:linux@brodo.de)>
- Rafael J. Wysocki <[rafael.j.wysocki@intel.com](mailto:rafael.j.wysocki@intel.com)>
- Viresh Kumar <[viresh.kumar@linaro.org](mailto:viresh.kumar@linaro.org)>

## 1. 怎么做?

如果, 你刚刚得到了一个全新的 CPU/芯片组及其数据手册, 并希望为这个 CPU/芯片组添加 cpufreq 支持? 很好, 这里有一些至关重要的提示:

### 1.1 初始化

首先, 在 \_\_initcall level 7 (module\_init()) 或更靠后的函数中检查这个内核是否运行在正确的 CPU 和正确的芯片组上。如果是, 则使用 cpufreq\_register\_driver() 向 CPUfreq 核心层注册一个 cpufreq\_driver 结构体。

结构体 cpufreq\_driver 应该包含什么成员?

- .name - 驱动的名字。
- .init - 一个指向 per-policy 初始化函数的指针。
- .verify - 一个指向” verification” 函数的指针。
- .setpolicy 或.fast\_switch 或.target 或.target\_index - 差异见下文。

其它可选成员

- .flags - 给 cpufreq 核心的提示。
- .driver\_data - cpufreq 驱动程序的特有数据。
- .get\_intermediate 和 target\_intermediate - 用于在改变 CPU 频率时切换到稳定的频率。
- .get - 返回 CPU 的当前频率。
- .bios\_limit - 返回 HW/BIOS 对 CPU 的最大频率限制值。
- .exit - 一个指向 per-policy 清理函数的指针, 该函数在 CPU 热插拔过程的 CPU\_POST\_DEAD 阶段被调用。
- .suspend - 一个指向 per-policy 暂停函数的指针, 该函数在关中断且在该策略的调节器停止后被调用。
- .resume - 一个指向 per-policy 恢复函数的指针, 该函数在关中断且在调节器再一次启动前被调用。

- .ready - 一个指向 per-policy 准备函数的指针，该函数在策略完全初始化之后被调用。
- .attr - 一个指向 NULL 结尾的” struct freq\_attr” 列表的指针，该列表允许导出值到 sysfs。
- .boost\_enabled - 如果设置，则启用提升 (boost) 频率。
- .set\_boost - 一个指向 per-policy 函数的指针，该函数用来开启/关闭提升 (boost) 频率功能。

## 1.2 Per-CPU 初始化

每当一个新的 CPU 被注册到设备模型中，或者当 cpufreq 驱动注册自身之后，如果此 CPU 的 cpufreq 策略不存在，则会调用 per-policy 的初始化函数 cpufreq\_driver.init。请注意，.init() 和.exit() 例程只为某个策略调用一次，而不是对该策略管理的每个 CPU 调用一次。它需要一个 struct cpufreq\_policy \*policy 作为参数。现在该怎么做呢？

如果有必要，请在你的 CPU 上激活 CPUfreq 功能支持。

然后，驱动程序必须填写以下值：

policy->cpuinfo.min_freq 和 policy->cpuinfo.max_freq	该 CPU 支持的最低和最高频率 (kHz)
policy->cpuinfo.transition_latency	CPU 在两个频率之间切换所需的时间，以纳秒为单位（如不适用，设定为 CPUFREQ_ETERNAL）
policy->cur	该 CPU 当前的工作频率 (如适用)
policy->min, policy->max, policy->policy_and, if necessary, policy->governor	必须包含该 CPU 的”默认策略”。稍后会用这些值调用 cpufreq_driver.verify 和下面函数之一：cpufreq_driver.setpolicy 或 cpufreq_driver.target/target_index
policy->cpus	该 policy 通过 DVFS 框架影响的全部 CPU (即与本 CPU 共享”时钟/电压”对) 构成掩码 (同时包含在线和离线 CPU)，用掩码更新本字段

对于设置其中的一些值 (cpuinfo.min/max\_freq, policy->min/max)，频率表辅助函数可能会有帮助。关于它们的更多信息，请参见第 2 节。

## 1.3 验证

当用户决定设置一个新的策略 (由” policy,governor,min,max 组成” ) 时，必须对这个策略进行验证，以便纠正不兼容的值。为了验证这些值，cpufreq\_verify\_within\_limits(struct cpufreq\_policy \*policy, unsigned int min\_freq, unsigned int max\_freq) 函数可能会有帮助。关于频率表辅助函数的详细内容请参见第 2 节。

您需要确保至少有一个有效频率（或工作范围）在 policy->min 和 policy->max 范围内。如果有必要，先增大 policy->max，只有在没有解决方案的情况下，才减小 policy->min。

### 1.4 target 或 target\_index 或 setpolicy 或 fast\_switch?

大多数 cpufreq 驱动甚至大多数 CPU 频率升降算法只允许将 CPU 频率设置为预定义的固定值。对于这些，你可以使用->target(), ->target\_index() 或->fast\_switch() 回调。

有些具有硬件调频能力的处理器可以自行依据某些限制来切换 CPU 频率。它们应使用->setpolicy() 回调。

### 1.5. target/target\_index

target\_index 调用有两个参数: `struct cpufreq_policy * policy` 和 `unsigned int` 索引 (用于索引频率表项)。

当调用这里时, CPUfreq 驱动必须设置新的频率。实际频率必须由 `freq_table[index].frequency` 决定。

在发生错误的情况下总是应该恢复到之前的频率 (即 `policy->restore_freq`), 即使我们已经切换到了中间频率。

### 已弃用

target 调用有三个参数。`struct cpufreq_policy * policy`, `unsigned int target_frequency`, `unsigned int relation`.

CPUfreq 驱动在调用这里时必须设置新的频率。实际的频率必须使用以下规则来确定。

- 尽量贴近”目标频率”。
- `policy->min <= new_freq <= policy->max` (这必须是有效的!!!)
- 如果 `relation==CPUFREQ_REL_L`, 尝试选择一个高于或等于 `target_freq` 的 `new_freq`。 (“L 代表最低, 但不能低于” )
- 如果 `relation==CPUFREQ_REL_H`, 尝试选择一个低于或等于 `target_freq` 的 `new_freq`。 (“H 代表最高, 但不能高于” )

这里, 频率表辅助函数可能会帮助你 - 详见第 2 节。

### 1.6. fast\_switch

这个函数用于从调度器的上下文进行频率切换。并非所有的驱动都要实现它, 因为不允许在这个回调中睡眠。这个回调必须经过高度优化, 以尽可能快地进行切换。

这个函数有两个参数: `struct cpufreq_policy *policy` 和 `unsigned int target_frequency`。

## 1.7 setpolicy

setpolicy 调用只需要一个 `struct cpufreq_policy * policy` 作为参数。需要将处理器内或芯片组内动态频率切换的下限设置为 `policy->min`, 上限设置为 `policy->max`, 如果支持的话, 当 `policy->policy` 为 `CPUFREQ_POLICY_PERFORMANCE` 时选择面向性能的设置, 为 `CPUFREQ_POLICY_POWERSAVE` 时选择面向省电的设置。也可以查看 `drivers/cpufreq/longrun.c` 中的参考实现。

## 1.8 get\_intermediate 和 target\_intermediate

仅适用于未设置 `target_index()` 和 `CPUFREQ_ASYNC_NOTIFICATION` 的驱动。

`get_intermediate` 应该返回一个平台想要切换到的稳定的中间频率, `target_intermediate()` 应该将 CPU 设置为该频率, 然后再跳转到' index' 对应的频率。`cpufreq` 核心会负责发送通知, 驱动不必在 `target_intermediate()` 或 `target_index()` 中处理它们。

在驱动程序不想为某个目标频率切换到中间频率的情况下, 它们可以让 `get_intermediate()` 返回' 0'。在这种情况下, `cpufreq` 核心将直接调用`->target_index()`。

注意: `->target_index()` 应该在发生失败的情况下将频率恢复到 `policy->restore_freq`, 因为 `cpufreq` 核心会为此发送通知。

## 2. 频率表辅助函数

由于大多数支持 `cpufreq` 的处理器只允许被设置为几个特定的频率, 因此, ”频率表” 和一些相关函数可能会辅助处理器驱动程序的一些工作。这样的” 频率表” 是一个由 `struct cpufreq_frequency_table` 的条目构成的数组,” driver\_data” 成员包含驱动程序的专用值,” frequency” 成员包含了相应的频率, 此外还有标志成员。在表的最后, 需要添加一个 `cpufreq_frequency_table` 条目, 频率设置为 `CPUFREQ_TABLE_END`。如果想跳过表中的一个条目, 则将频率设置为 `CPUFREQ_ENTRY_INVALID`。这些条目不需要按照任何特定的顺序排序, 如果排序了, `cpufreq` 核心执行 DVFS 会更快一点, 因为搜索最佳匹配会更快。

如果在 `policy->freq_table` 字段中包含一个有效的频率表指针, 频率表就会被 `cpufreq` 核心自动验证。

`cpufreq_frequency_table_verify()` 保证至少有一个有效的频率在 `policy->min` 和 `policy->max` 范围内, 并且所有其他准则都被满足。这对`->verify` 调用很有帮助。

`cpufreq_frequency_table_target()` 是对应于`->target` 阶段的频率表辅助函数。只要把值传递给这个函数, 这个函数就会返回包含 CPU 要设置的频率的频率表条目。

以下宏可以作为 `cpufreq_frequency_table` 的迭代器。

`cpufreq_for_each_entry(pos, table)` - 遍历频率表的所有条目。

`cpufreq_for_each_valid_entry(pos, table)` - 该函数遍历所有条目, 不包括 `CPUFREQ_ENTRY_INVALID` 频率。使用参数” pos” - 一个 `cpufreq_frequency_table *` 作为循环指针, 使用参数” table” - 作为你想迭代的 `cpufreq_frequency_table *`。

例如:

```
struct cpufreq_frequency_table *pos, *driver_freq_table;  
  
cpufreq_for_each_entry(pos, driver_freq_table) {  
    /* Do something with pos */  
    pos->frequency = ...  
}
```

如果你需要在 `driver_freq_table` 中处理 `pos` 的位置，不要做指针减法，因为它的代价相当高。作为替代，使用宏 `cpufreq_for_each_entry_idx()` 和 `cpufreq_for_each_valid_entry_idx()`。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:<alexs@kernel.org>)。

**Original** Documentation/cpu-freq/cpufreq-stats.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 唐艺舟 Tang Yizhou <[tangyeechou@gmail.com](mailto:tangyeechou@gmail.com)>

### sysfs CPUFreq Stats 的一般说明

为使用者准备的信息

作者: Venkatesh Pallipadi <[venkatesh.pallipadi@intel.com](mailto:venkatesh.pallipadi@intel.com)>

## 1. 简介

`cpufreq-stats` 是一种为每个 CPU 提供 CPU 频率统计的驱动。这些统计数据以`/sysfs` 中一系列只读接口的形式呈现。`cpufreq-stats` 接口（若已配置）将为每个 CPU 生成 `/sysfs (<sysfs root>/devices/system/cpu/cpuX/cpufreq/stats/)` 中 `cpufreq` 目录下的 `stats` 目录。各项统计数据将在 `stats` 目录下形成对应的只读文件。

此驱动是以独立于任何可能运行在你所用 CPU 上的特定 `cpufreq_driver` 的方式设计的。因此，它将能和任何 `cpufreq_driver` 协同工作。

## 2. 已提供的统计数据 (有例子)

cpufreq stats 提供了以下统计数据（在下面详细解释）。

- time\_in\_state
- total\_trans
- trans\_table

所有统计数据来自以下时间范围：从统计驱动被加载的时间（或统计数据被重置的时间）开始，到某一统计数据被读取的时间为止。显然，统计驱动不会保存它被加载之前的任何频率转换信息。

```
<mysystem>:/sys/devices/system/cpu/cpu0/cpufreq/stats # ls -l
total 0
drwxr-xr-x 2 root root 0 May 14 16:06 .
drwxr-xr-x 3 root root 0 May 14 15:58 ..
--w----- 1 root root 4096 May 14 16:06 reset
-r--r--r-- 1 root root 4096 May 14 16:06 time_in_state
-r--r--r-- 1 root root 4096 May 14 16:06 total_trans
-r--r--r-- 1 root root 4096 May 14 16:06 trans_table
```

- **reset**

只写属性，可用于重置统计计数器。这对于评估不同调节器的系统行为非常有用，且无需重启。

- **time\_in\_state**

此文件给出了在本 CPU 支持的每个频率上分别花费的时间。cat 输出的每一行都是一个”`<frequency><time>`”对，表示这个 CPU 在 `<frequency>` 上花费了 `<time>` 个 usertime 单位的时间。输出的每一行对应一个 CPU 支持的频率。这里 usertime 单位是 10mS（类似于/proc 导出的其它时间）。

```
<mysystem>:/sys/devices/system/cpu/cpu0/cpufreq/stats # cat time_in_state
3600000 2089
3400000 136
3200000 34
3000000 67
2800000 172488
```

- **total\_trans**

此文件给出了这个 CPU 频率转换的总次数。cat 的输出是一个计数值，它就是频率转换的总次数。

```
<mysystem>:/sys/devices/system/cpu/cpu0/cpufreq/stats # cat total_trans
20
```

- **trans\_table**

本文件提供所有 CPU 频率转换的细粒度信息。这里的 cat 输出是一个二维矩阵，其中一个条目 `<i, j>`（第 i 行，第 j 列）代表从 Freq\_i 到 Freq\_j 的转换次数。Freq\_i 行和 Freq\_j 列遵循驱动最初提供给 cpufreq 核心的频率表的排列顺序，因此可以已排序（升序或降序）或未排序。这里的输出也包含了实际频率值，分别按行和按列显示，以便更好地阅读。

如果转换表大于 PAGE\_SIZE，读取时将返回一个-EFBIG 错误。

```
<mysystem>:/sys/devices/system/cpu/cpu0/cpufreq/stats # cat trans_table
From : To
      : 3600000 3400000 3200000 3000000 2800000
3600000:   0     5     0     0     0
3400000:   4     0     2     0     0
3200000:   0     1     0     2     0
3000000:   0     0     1     0     3
2800000:   0     0     0     2     0
```

### 3. 配置 cpufreq-stats

按以下方式在你的内核中配置 cpufreq-stats:

```
Config Main Menu
  Power management options (ACPI, APM) --->
    CPU Frequency scaling --->
      [*] CPU Frequency scaling
      [*] CPU frequency translation statistics
```

“CPU Frequency scaling” (CONFIG\_CPU\_FREQ) 应该被启用，以支持配置 cpufreq-stats。

“CPU frequency translation statistics” (CONFIG\_CPU\_FREQ\_STAT) 提供了包括 time\_in\_state、total\_trans 和 trans\_table 的统计数据。

一旦启用了这个选项，并且你的 CPU 支持 cpufreq，你就可以在/sysfs 中看到 CPU 频率统计。

### 邮件列表

这里有一个 CPU 频率变化的 CVS 提交和通用列表，您可以在这里报告 bug、问题或提交补丁。要发布消息，请发送电子邮件到 [linux-pm@vger.kernel.org](mailto:linux-pm@vger.kernel.org)。

### 链接

FTP 档案: \* <ftp://ftp.linux.org.uk/pub/linux/cpufreq/>

如何访问 CVS 仓库: \* <http://cvs.arm.linux.org.uk/>

CPUFreq 邮件列表: \* <http://vger.kernel.org/vger-lists.html#linux-pm>

SA-1100 的时钟和电压标度: \* <http://www.lartmaker.nl/projects/scaling>

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题, 请联系该文件的译者, 或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/iio/index.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## \* 工业 I/O

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解, 而不是作为一个分支。因此, 如果您对此文件有任何意见或更新, 请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题, 请联系该文件的译者, 或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/iio/iio\_configfs.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## 工业 IIO configfs 支持

### 1. 概述

Configfs 是一种内核对象的基于文件系统的管理系统, IIO 使用一些可以通过 configfs 轻松配置的对象 (例如: 设备, 触发器)。

关于 configfs 是如何运行的, 请查阅 Documentation/filesystems/configfs.rst 了解更多信息。

### 2. 用法

为了使 configfs 支持 IIO, 我们需要在编译时选中 config 的 CONFIG\_IIO\_CONFIGFS 选项。

然后, 挂载 configfs 文件系统 (通常在 /config directory 目录下)::

```
$ mkdir/config $ mount -t configfs none/config
```

此时, 将创建所有默认 IIO 组, 并可以在/ config / iio 下对其进行访问。下一章将介绍可用的 IIO 配置对象。

### 3. 软件触发器

IIO 默认 configfs 组之一是“触发器”组。挂载 configfs 后可以自动访问它，并且可以在/config/iio/triggers 下找到。

IIO 软件触发器为创建多种触发器类型提供了支持。通常在 include/linux/iio/sw\_trigger.h: 中的接口下将新的触发器类型实现为单独的内核模块：

```
/*
 * drivers/iio/trigger/iio-trig-sample.c
 * 一种新触发器类型的内核模块实例
 */
#include <linux/iio/sw_trigger.h>

static struct iio_sw_trigger *iio_trig_sample_probe(const char *name)
{
    /*
     * 这将分配并注册一个 IIO 触发器以及其他触发器类型特性的初始化。
     */
}

static int iio_trig_sample_remove(struct iio_sw_trigger *swt)
{
    /*
     * 这会废弃 iio_trig_sample_probe 中的操作
     */
}

static const struct iio_sw_trigger_ops iio_trig_sample_ops = {
    .probe      = iio_trig_sample_probe,
    .remove     = iio_trig_sample_remove,
};

static struct iio_sw_trigger_type iio_trig_sample = {
    .name = "trig-sample",
    .owner = THIS_MODULE,
    .ops = &iio_trig_sample_ops,
};
```

module\_iio\_sw\_trigger\_driver(iio\_trig\_sample);

每种触发器类型在/config/iio/triggers 下都有其自己的目录。加载 iio-trig-sample 模块将创建“trig-sample”触发器类型目录/config/iio/triggers/trig-sample.

我们支持以下中断源（触发器类型）

- hrtimer, 使用高分辨率定时器作为中断源

### 3.1 Hrtimer 触发器创建与销毁

加载 iio-trig-hrtimer 模块将注册 hrtimer 触发器类型，从而允许用户在 /config/iio/triggers/hrtimer 下创建 hrtimer 触发器。

例如：

```
$ mkdir /config/iio/triggers/hrtimer/instance1
$ rmdir /config/iio/triggers/hrtimer/instance1
```

每个触发器可以具有一个或多个独特的触发器类型的属性。

### 3.2 “hrtimer” 触发器类型属性

“hrtimer” 触发器类型没有来自/config dir 的任何可配置属性。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/iio/ep93xx\_adc.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## 思睿逻辑 EP93xx 模拟数字转换器驱动

### 1. 概述

该驱动同时适用于具有 5 通道模拟数字转换器的低端 (EP9301, Ep9302) 设备和 10 通道触摸屏/模拟数字转换器的高端设备 (EP9307, EP9312, EP9315)。

### 2. 通道编号

EP9301 和 EP9302 数据表定义了通道 0..4 的编号方案。虽然 EP9307, EP9312 和 EP9315 多了 3 个通道（一共 8 个），但是编号并没有定义。所以说最后三个通道是随机编号的。

如果 ep93xx\_adc 是 IIO 设备 0，您将在以下位置找到条目 /sys/bus/iio/devices/iio:device0/：

sysfs 入口	ball/pin 名称
in_voltage0_raw	YM
in_voltage1_raw	SXP
in_voltage2_raw	SXM
in_voltage3_raw	SYP
in_voltage4_raw	SYM
in_voltage5_raw	XP
in_voltage6_raw	XM
in_voltage7_raw	YP

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:<alexs@kernel.org>)。

---

### Original Documentation/infiniband/index.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 王普宇 Puyu Wang <[realpuyuwang@gmail.com](mailto:realpuyuwang@gmail.com)> 时奎亮 Alex Shi  
<[alexs@kernel.org](mailto:<alexs@kernel.org>)>

### \* **infiniband**

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:<alexs@kernel.org>)。

---

### Original Documentation/infiniband/core\_locking.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 王普宇 Puyu Wang <[realpuyuwang@gmail.com](mailto:realpuyuwang@gmail.com)> 时奎亮 Alex Shi  
<[alexs@kernel.org](mailto:<alexs@kernel.org>)>

## infiniband 中间层锁

本指南试图明确 infiniband 中间层的锁假设。它描述了对位于中间层以下的低级驱动程序和使用中间层的上层协议的要求。

## 睡眠和中断环境

除了以下异常情况，`ib_device` 结构体中所有方法的低级驱动实现都可以睡眠。这些异常情况是列表中的任意的方法：

- `create_ah`
- `modify_ah`
- `query_ah`
- `destroy_ah`
- `post_send`
- `post_recv`
- `poll_cq`
- `req_notify_cq`

他们可能不可以睡眠，而且必须可以从任何上下文中调用。

向上层协议使用者输出的相应函数：

- `rdma_create_ah`
- `rdma_modify_ah`
- `rdma_query_ah`
- `rdma_destroy_ah`
- `ib_post_send`
- `ib_post_recv`
- `ib_req_notify_cq`

因此，在任何情况下都可以安全调用（它们）。

此外，该函数

- `ib_dispatch_event`

被底层驱动用来通过中间层调度异步事件的“A”，也可以从任何上下文中安全调用。

## 可重入性

由低级驱动程序导出的 `ib_device` 结构体中的所有方法必须是完全可重入的。即使使用同一对象的多个函数调用被同时运行，低级驱动程序也需要执行所有必要的同步以保持一致性。

IB 中间层不执行任何函数调用的序列化。

因为低级驱动程序是可重入的，所以不要求上层协议使用者任何顺序执行。然而，为了得到合理的结果，可能需要一些顺序。例如，一个使用者可以在多个 CPU 上同时安全地调用 `ib_poll_cq()`。然而，不同的 `ib_poll_cq()` 调用之间的工作完成信息的顺序没有被定义。

## 回调

低级驱动程序不得直接从与 `ib_device` 方法调用相同的调用链中执行回调。例如，低级驱动程序不允许从 `post_send` 方法直接调用使用者的完成事件处理程序。相反，低级驱动程序应该推迟这个回调，例如，调度一个 tasklet 来执行这个回调。

低层驱动负责确保同一 CQ 的多个完成事件处理程序不被同时调用。驱动程序必须保证一个给定的 CQ 的事件处理程序在同一时间只有一个在运行。换句话说，以下情况是不允许的：

CPU1 <pre>low-level driver -&gt; consumer CQ event callback: /* ... */ ib_req_notify_cq(cq, ...);  /* ... */  return from CQ event handler</pre>	CPU2 <pre>low-level driver -&gt; consumer CQ event callback: /* ... */</pre>
---	---

完成事件和异步事件回调的运行环境没有被定义。根据低级别的驱动程序，它可能是进程上下文、softirq 上下文或中断上下文。上层协议使用者可能不会在回调中睡眠。

## 热插拔

当一个低级驱动程序调用 `ib_register_device()` 时，它宣布一个设备已经准备好供使用者使用，所有的初始化必须在这个调用之前完成。设备必须保持可用，直到驱动对 `ib_unregister_device()` 的调用返回。

低级驱动程序必须从进程上下文调用 `ib_register_device()` 和 `ib_unregister_device()`。如果使用者在这些调用中回调到驱动程序，它不能持有任何可能导致死锁的 semaphores。

一旦其结构体 `ib_client` 的 `add` 方法被调用，上层协议使用者就可以开始使用一个 IB 设备。使用者必须在从移除方法返回之前完成所有的清理工作并释放与设备相关的所有资源。

使用者被允许在其添加和删除方法中睡眠。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

### Original Documentation/infiniband/ipoib.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 王 普 宇 Puyu Wang <[realpuyuwang@gmail.com](mailto:realpuyuwang@gmail.com)> 时 奎 亮 Alex Shi <[alexs@kernel.org](mailto:alexs@kernel.org)>

## infiniband 上的 IP (IPoIB)

ib\_ipoib 驱动是 IETF ipoib 工作组发布的 RFC 4391 和 4392 所规定的 infiniband 上 IP 协议的一个实现。它是一个“本地”实现，即把接口类型设置为 ARPHRD\_INFINIBAND，硬件地址长度为 20（早期的专有实现向内核伪装为以太网接口）。

## 分区和 P\_Keys

当 IPoIB 驱动被加载时，它会使用索引为 0 的 P\_Key 给每个端口创建一个接口。要用不同的 P\_Key 创建一个接口，将所需的 P\_Key 写入主接口的 /sys/class/net/<intf name>/create\_child 文件里面。比如说：

```
echo 0x8001 > /sys/class/net/ib0/create_child
```

这将用 P\_Key 0x8001 创建一个名为 ib0.8001 的接口。要删除一个子接口，使用 delete\_child 文件：

```
echo 0x8001 > /sys/class/net/ib0/delete_child
```

任何接口的 P\_Key 都由“pkey”文件给出，而子接口的主接口在“parent”中。

子接口的创建/删除也可以使用 IPoIB 的 rtnl\_link\_ops 来完成，使用两种方式创建的子接口的行为是一样的。

### 数据报与连接模式

IPoIB 驱动支持两种操作模式：数据报和连接。模式是通过接口的 `/sys/class/net/<intf name>/mode` 文件设置和读取的。

在数据报模式下，使用 IB UD（不可靠数据报）传输，因此接口 MTU 等于 IB L2 MTU 减去 IPoIB 封装头（4 字节）。例如，在一个典型的具有 2K MTU 的 IB 结构中，IPoIB MTU 将是  $2048 - 4 = 2044$  字节。

在连接模式下，使用 IB RC（可靠的连接）传输。连接模式利用 IB 传输的连接特性，允许 MTU 达到最大的 IP 包大小 64K，这减少了处理大型 UDP 数据包、TCP 段等所需的 IP 包数量，提高了大型信息的性能。

在连接模式下，接口的 UD QP 仍被用于组播和与不支持连接模式的对等体的通信。在这种情况下，ICMP PMTU 数据包的 RX 仿真被用来使网络堆栈对这些邻居使用较小的 UD MTU。

### 无状态卸载

如果 IB HW 支持 IPoIB 无状态卸载，IPoIB 会向网络堆栈广播 TCP/IP 校验和/或大量传送 (LSO) 负载转移能力。

大量传送 (LSO) 负载转移也已实现，可以使用 ethtool 调用打开/关闭。目前，LRO 只支持具有校验和卸载能力的设备。

无状态卸载只在数据报模式下支持。

### 中断管理

如果底层 IB 设备支持 CQ 事件管理，可以使用 ethtool 来设置中断缓解参数，从而减少处理中断产生的开销。IPoIB 的主要代码路径不使用 TX 完成信号的事件，所以只支持 RX 管理。

### 调试信息

通过将 `CONFIG_INFINIBAND_IPOIB_DEBUG` 设置为 “y” 来编译 IPoIB 驱动，跟踪信息被编译到驱动中。通过将模块参数 `debug_level` 和 `mcast_debug_level` 设置为 1 来打开它们。这些参数可以在运行时通过 `/sys/module/ib_ipoib/` 的文件来控制。

`CONFIG_INFINIBAND_IPOIB_DEBUG` 也启用 debugfs 虚拟文件系统中的文件。通过挂载这个文件系统，例如用：

```
mount -t debugfs none /sys/kernel/debug
```

可以从 `/sys/kernel/debug/ipoib/ib0_mcgr` 等文件中获得关于多播组的统计数据。

这个选项对性能的影响可以忽略不计，所以在正常运行时，在 `debug_level` 设置为 0 的情况下启用这个选项是安全的。

`CONFIG_INFINIBAND_IPOIB_DEBUG_DATA` 当 `data_debug_level` 设置为 1 时，可以在数据路径中启用更多的调试输出。然而，即使禁用输出，启用这个配置选项也会影响性能，因为它在快速路径中增加了测试。

## 引用

在 InfiniBand 上传输 IP (IPoIB) (RFC 4391)。<http://ietf.org/rfc/rfc4391.txt>

infiniband 上的 IP: 上的 IP 架构 (RFC 4392)。<http://ietf.org/rfc/rfc4392.txt>

infiniband 上的 IP: 连接模式 (RFC 4755) <http://ietf.org/rfc/rfc4755.txt>

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

**Original** Documentation/infiniband/opa\_vnic.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

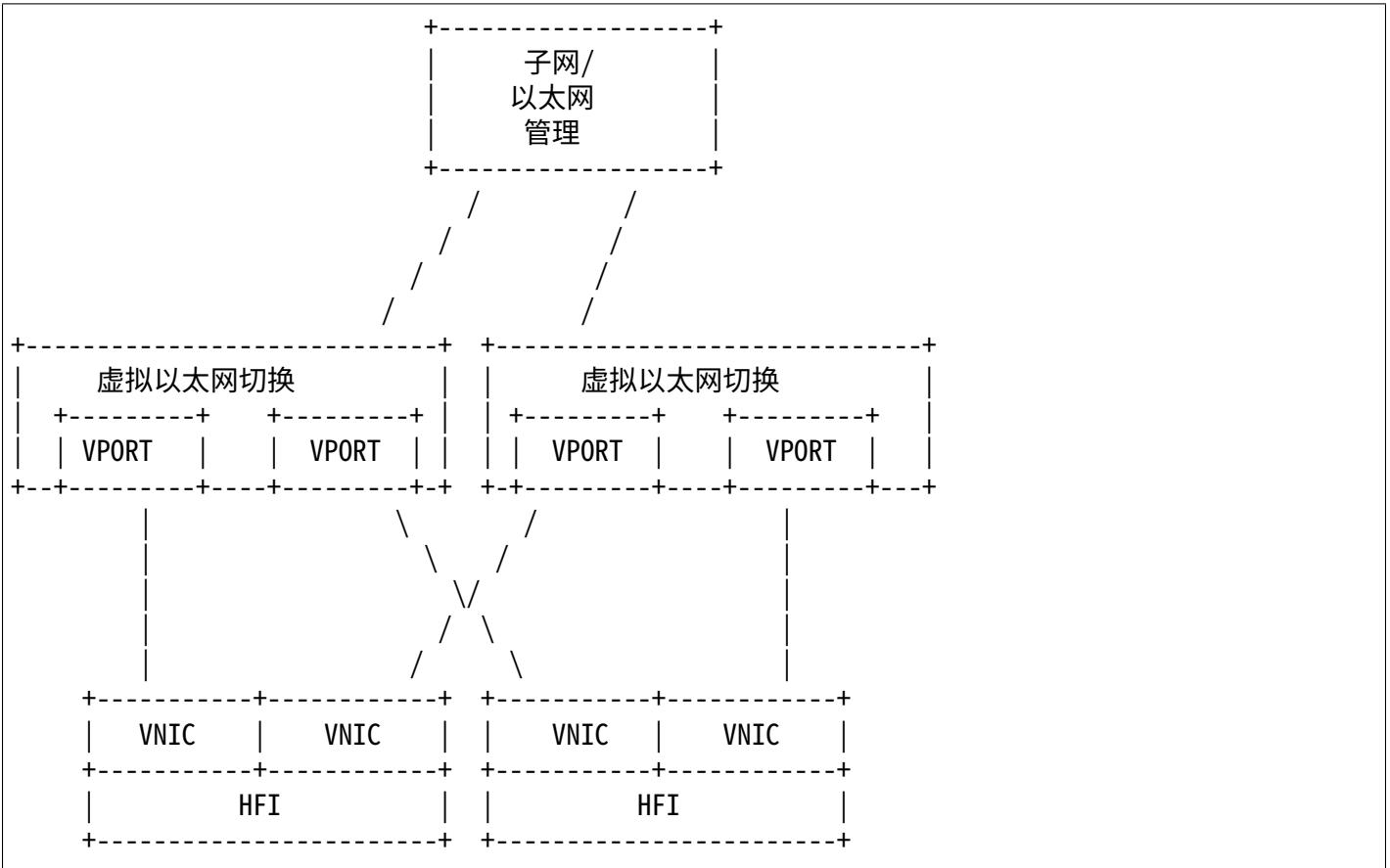
校译 王 普 宇 Puyu Wang <[realpuyuwang@gmail.com](mailto:realpuyuwang@gmail.com)> 时 奎 亮 Alex Shi  
<[alexs@kernel.org](mailto:alexs@kernel.org)>

## 英特尔全路径 (OPA) 虚拟网络接口控制器 (VNIC)

英特尔全路径 (OPA) 虚拟网络接口控制器 (VNIC) 功能通过封装 HFI 节点之间的以太网数据包，支持 Omni-Path 结构上的以太网功能。

## 体系结构

Omni-Path 封装的以太网数据包的交换模式涉及 Omni-Path 结构拓扑上覆盖的一个或多个虚拟以太网交换机。Omni-Path 结构上的 HFI 节点的一个子集被允许在特定的虚拟以太网交换机上交换封装的以太网数据包。虚拟以太网交换机是通过配置结构上的 HFI 节点实现的逻辑抽象，用于生成和处理报头。在最简单的配置中，整个结构的所有 HFI 节点通过一个虚拟以太网交换机交换封装的以太网数据包。一个虚拟以太网交换机，实际上是一个独立的以太网网络。该配置由以太网管理器 (EM) 执行，它是可信的结构管理器 (FM) 应用程序的一部分。HFI 节点可以有多个 VNIC，每个连接到不同的虚拟以太网交换机。下图介绍了两个虚拟以太网交换机与两个 HFI 节点的情况：



Omni-Path 封装的以太网数据包格式如下所述。

位	域
Quad Word 0:	
0-19	SLID (低 20 位)
20-30	长度 (以四字为单位)
31	BECN 位
32-51	DLID (低 20 位)
52-56	SC (服务级别)
57-59	RC (路由控制)
60	FECN 位
61-62	L2 (=10, 16B 格式)
63	LT (=1, 链路传输头 Flit)
Quad Word 1:	
0-7	L4 type (=0x78 ETHERNET)
8-11	SLID[23:20]
12-15	DLID[23:20]
16-31	PKEY
32-47	熵
48-63	保留
Quad Word 2:	
0-15	保留
16-31	L4 头
32-63	以太网数据包
Quad Words 3 to N-1:	
0-63	以太网数据包 (pad 拓展)
Quad Word N (last):	
0-23	以太网数据包 (pad 拓展)
24-55	ICRC
56-61	尾
62-63	LT (=01, 链路传输尾 Flit)

以太网数据包在传输端被填充，以确保 VNIC OPA 数据包是四字对齐的。“尾”字段包含填充的字节数。在接收端，“尾”字段被读取，在将数据包向上传递到网络堆栈之前，填充物被移除（与 ICRC、尾和 OPA 头一起）。

L4 头字段包含 VNIC 端口所属的虚拟以太网交换机 ID。在接收端，该字段用于将收到的 VNIC 数据包去多路复用到不同的 VNIC 端口。

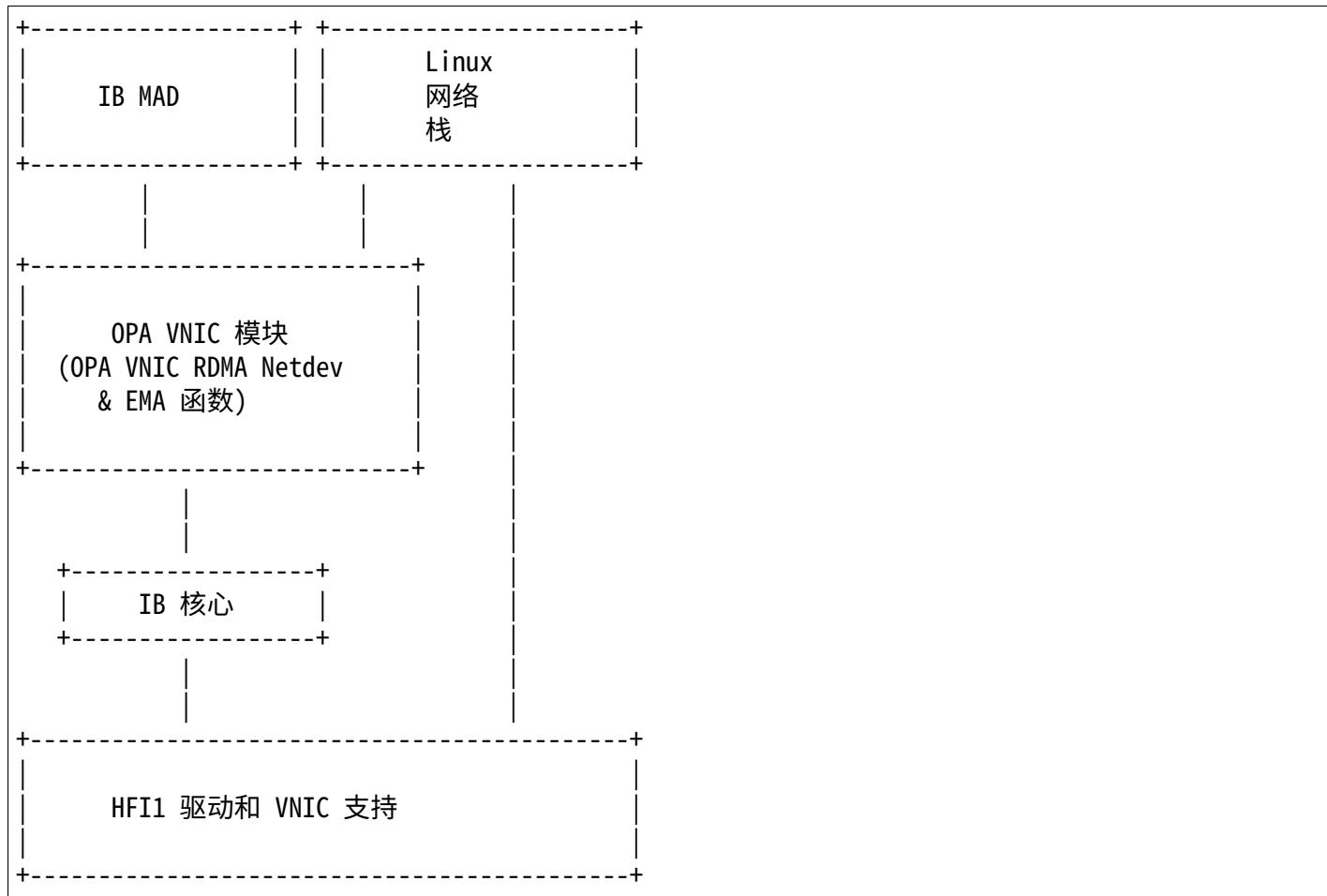
## 驱动设计

英特尔 OPA VNIC 的软件设计如下图所示。OPA VNIC 功能有一个依赖于硬件的部分和一个独立于硬件的部分。

对 IB 设备分配和释放 RDMA netdev 设备的支持已经被加入。RDMA netdev 支持与网络堆栈的对接，从而创建标准的网络接口。OPA\_VNIC 是一个 RDMA netdev 设备类型。

依赖于 HW 的 VNIC 功能是 HFI1 驱动的一部分。它实现了分配和释放 OPA\_VNIC RDMA netdev 的动作。它涉及 VNIC 功能的 HW 资源分配/管理。它与网络堆栈接口并实现所需的 `net_device_ops` 功能。它在传输路径中期待 Omni-Path 封装的以太网数据包，并提供对它们的 HW 访问。在将数据包向上传递到网络堆栈之前，它把 Omni-Path 头从接收的数据包中剥离。它还实现了 RDMA netdev 控制操作。

OPA VNIC 模块实现了独立于硬件的 VNIC 功能。它由两部分组成。VNIC 以太网管理代理 (VEMA) 作为一个 IB 客户端向 IB 核心注册，并与 IB MAD 栈接口。它与以太网管理器 (EM) 和 VNIC netdev 交换管理信息。VNIC netdev 部分分配和释放 OPA\_VNIC RDMA netdev 设备。它在需要时覆盖由依赖 HW 的 VNIC 驱动设置的 `net_device_ops` 函数，以适应任何控制操作。它还处理以太网数据包的封装，在传输路径中使用 Omni-Path 头。对于每个 VNIC 接口，封装所需的信息是由 EM 通过 VEMA MAD 接口配置的。它还通过调用 RDMA netdev 控制操作将任何控制信息传递给依赖于 HW 的驱动程序：



**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

### Original Documentation/infiniband/sysfs.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 王 普 宇 Puyu Wang <[realpuyuwang@gmail.com](mailto:realpuyuwang@gmail.com)> 时 奎 亮 Alex Shi <[alexs@kernel.org](mailto:alexs@kernel.org)>

## Sysfs 文件

sysfs 接口已移至 Documentation/ABI/stable/sysfs-class-infiniband.

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

### Original Documentation/infiniband/tag\_matching.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 王 普 宇 Puyu Wang <[realpuyuwang@gmail.com](mailto:realpuyuwang@gmail.com)> 时 奎 亮 Alex Shi <[alexs@kernel.org](mailto:alexs@kernel.org)>

## 标签匹配逻辑

MPI 标准定义了一套规则，称为标签匹配，用于将源发送操作与目的接收匹配。以下参数必须与以下源和目的参数相匹配：

- 沟通者
- 用户标签-通配符 (wild card) 可由接收方指定
- 来源等级-通配符可由接收方指定

- 目的地等级 - wild

排序规则要求，当一对以上的发送和接收消息信封可能匹配时，包括最早发布-发送和最早发布-接收的一对是必须用来满足匹配操作的一对。然而，这并不意味着标签是按照它们被创建的顺序消耗的，例如，如果早期的标签不能用来满足匹配规则，那么后来生成的标签可能被消耗。

当消息从发送方发送到接收方时，通信库可能试图在相应的匹配接收被发布之后或之前处理该操作。如果匹配的接收被发布，这就是一个预期的消息，否则就被称为一个意外的消息。实现时经常为这两种不同的匹配实例使用不同的匹配方案。

为了减少 MPI 库的内存占用，MPI 实现通常使用两种不同的协议来实现这一目的：

1. Eager 协议-当发送方处理完发送时，完整的信息就会被发送。在 `send_cq` 中会收到一个完成发送的通知，通知缓冲区可以被重新使用。
2. Rendezvous 协议-发送方在第一次通知接收方时发送标签匹配头，也许还有一部分数据。当相应的缓冲区被发布时，响应者将使用头中的信息，直接向匹配的缓冲区发起 RDMA 读取操作。为了使缓冲区得到重用，需要收到一个 `fin` 消息。

### 标签匹配的实现

使用的匹配对象有两种类型，即发布的接收列表和意外消息列表。应用程序通过调用发布的接收列表中的 MPI 接收例程发布接收缓冲区，并使用 MPI 发送例程发布发送消息。发布的接收列表的头部可以由硬件来维护，而软件则要对这个列表进行跟踪。

当发送开始并到达接收端时，如果没有为这个到达的消息预先发布接收，它将被传递给软件并被放在意外 (`unexpect`) 消息列表中。否则，将对该匹配进行处理，包括交会处理，如果合适的话，将数据传送到指定的接收缓冲区。这允许接收方 MPI 标签匹配与计算重叠。

当一个接收信息被发布时，通信库将首先检查软件的意外信息列表，以寻找一个匹配的接收信息。如果找到一个匹配的，数据就会被送到用户缓冲区，使用一个软件控制的协议。UCX 的实现根据数据大小，使用急切或交会协议。如果没有找到匹配，整个预置的接收列表由硬件维护，并且有空间在这个列表中增加一个预置的接收，这个接收被传递给硬件。软件要对这个列表进行跟踪，以帮助处理 MPI 取消操作。此外，由于硬件和软件在标签匹配操作方面预计不会紧密同步，这个影子列表被用来检测预先发布的接收被传递到硬件的情况，因为匹配的意外消息正在从硬件传递到软件。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/infiniband/user\_mad.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 王 普 宇 Puyu Wang <[realpuyuwang@gmail.com](mailto:realpuyuwang@gmail.com)> 时 奎 亮 Alex Shi <[alexs@kernel.org](mailto:alexs@kernel.org)>

## 用户空间 MAD 访问

### 设备文件

每个 InfiniBand 设备的每个端口都有一个“umad”设备和一个“issm”设备连接。例如，一个双端口的 HCA 将有两个 umad 设备和两个 issm 设备，而一个交换机将有每个类型的一个设备（对于交换机端口 0）。

### 创建 MAD 代理

一个 MAD 代理可以通过填写一个结构体 `ib_user_mad_reg_req` 来创建，然后在适当的设备文件的文件描述符上调用 `IB_USER_MAD_REGISTER_AGENT` ioctl。如果注册请求成功，结构体中会返回一个 32 位的 ID。比如说：

```
struct ib_user_mad_reg_req req = { /* ... */ };
ret = ioctl(fd, IB_USER_MAD_REGISTER_AGENT, (char *) &req);
if (!ret)
    my_agent = req.id;
else
    perror("agent register");
```

代理可以通过 `IB_USER_MAD_UNREGISTER_AGENT` ioctl 取消注册。另外，所有通过文件描述符注册的代理在描述符关闭时将被取消注册。

**2014** 现在提供了一个新的注册 IOCTL，允许在注册时提供额外的字段。这个注册调用的用户隐含了对 `pkey_index` 的使用（见下文）。现在提供了一个新的注册 IOCTL，允许在注册时提供额外的字段。这个注册调用的用户隐含了对 `pkey_index` 的使用（见下文）。

### 接收 MADs

使用 `read()` 接收 MAD。现在接收端支持 RMPP。传给 `read()` 的缓冲区必须至少是一个 `struct ib_user_mad + 256` 字节。比如说：

如果传递的缓冲区不足以容纳收到的 MAD（RMPP），`errno` 被设置为 `ENOSPC`，需要的缓冲区长度被设置在 `mad.length` 中。

正常 MAD(非 RMPP) 的读取示例：

```
struct ib_user_mad *mad;
mad = malloc(sizeof(*mad) + 256);
ret = read(fd, mad, sizeof(*mad) + 256);
```

```
if (ret != sizeof mad + 256) {
    perror("read");
    free(mad);
}
```

RMPP 读取示例:

```
struct ib_user_mad *mad;
mad = malloc(sizeof *mad + 256);
ret = read(fd, mad, sizeof *mad + 256);
if (ret == -ENOSPC) {
    length = mad.length;
    free(mad);
    mad = malloc(sizeof *mad + length);
    ret = read(fd, mad, sizeof *mad + length);
}
if (ret < 0) {
    perror("read");
    free(mad);
}
```

除了实际的 MAD 内容外，其他结构体 `ib_user_mad` 字段将被填入收到的 MAD 的信息。例如，远程 LID 将在 `mad.lid` 中。

如果发送超时，将产生一个接收，`mad.status` 设置为 `ETIMEDOUT`。否则，当一个 MAD 被成功接收后，`mad.status` 将是 0。

`poll()`/`select()` 可以用来等待一个 MAD 可以被读取。

`poll()`/`select()` 可以用来等待，直到可以读取一个 MAD。

## 发送 MADs

MADs 是用 `write()` 发送的。发送的代理 ID 应该填入 MAD 的 `id` 字段，目的地 LID 应该填入 `lid` 字段，以此类推。发送端确实支持 RMPP，所以可以发送任意长度的 MAD。比如说：

```
struct ib_user_mad *mad;

mad = malloc(sizeof *mad + mad_length);

/* fill in mad->data */

mad->hdr.id  = my_agent;          /* req.id from agent registration */
mad->hdr.lid = my_dest;           /* in network byte order... */
/* etc. */

ret = write(fd, &mad, sizeof *mad + mad_length);
if (ret != sizeof *mad + mad_length)
    perror("write");
```

## 交换 IDs

umad 设备的用户可以在发送的 MAD 中使用交换 ID 字段的低 32 位（也就是网络字节顺序中最小有效的一半字段）来匹配请求/响应对。上面的 32 位是保留给内核使用的，在发送 MAD 之前会被改写。

## P\_Key 索引处理

旧的 ib\_umad 接口不允许为发送的 MAD 设置 P\_Key 索引，也没有提供获取接收的 MAD 的 P\_Key 索引的方法。一个带有 pkey\_index 成员的 struct ib\_user\_mad\_hdr 的新布局已经被定义；然而，为了保持与旧的应用程序的二进制兼容性，除非在文件描述符被用于其他用途之前调用 IB\_USER\_MAD\_ENABLE\_PKEY 或 IB\_USER\_MAD\_REGISTER\_AGENT2 ioctl 之一，否则不会使用这种新布局。

在 2008 年 9 月，IB\_USER\_MAD\_ABI\_VERSION 将被增加到 6，默认使用新的 ib\_user\_mad\_hdr 结构布局，并且 IB\_USER\_MAD\_ENABLE\_PKEY ioctl 将被删除。

## 设置 IsSM 功能位

要为一个端口设置 IsSM 功能位，只需打开相应的 issm 设备文件。如果 IsSM 位已经被设置，那么打开调用将阻塞，直到该位被清除（或者如果 O\_NONBLOCK 标志被传递给 open()，则立即返回，errno 设置为 EAGAIN）。当 issm 文件被关闭时，IsSM 位将被清除。在 issm 文件上不能进行任何读、写或其他操作。

## /dev 文件

为了用 udev 自动创建相应的字符设备文件，一个类似：

```
KERNEL=="umad*", NAME="infiniband/%k"
KERNEL=="issm*", NAME="infiniband/%k"
```

的规则可以被使用。它将创建节点的名字::

```
/dev/infiniband/umad0
/dev/infiniband/issm0
```

为第一个端口，以此类推。与这些设备相关的 infiniband 设备和端口可以从以下文件中确定::

```
/sys/class/infiniband_mad/umad0/ibdev
/sys/class/infiniband_mad/umad0/port
```

和::

```
/sys/class/infiniband_mad/issm0/ibdev
/sys/class/infiniband_mad/issm0/port
```

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

**Original** Documentation/infiniband/user\_verbs.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 王 普 宇 Puyu Wang <[realpuyuwang@gmail.com](mailto:realpuyuwang@gmail.com)> 时 奎 亮 Alex Shi  
<[alexs@kernel.org](mailto:alexs@kernel.org)>

### 用户空间 verbs 访问

ib\_uverbs 模块，通过启用 CONFIG\_INFINIBAND\_USER\_VERBS 构建，使用户空间通过“verbs”直接访问 IB 硬件，如 InfiniBand 架构规范第 11 章所述。

要使用 verbs，需要 libibverbs 库，可从 <https://github.com/linux-rdma/rdma-core>。libibverbs 包含一个独立于设备的 API，用于使用 ib\_uverbs 接口。libibverbs 还需要为你的 InfiniBand 硬件提供适当的独立于设备的内核和用户空间驱动。例如，要使用 Mellanox HCA，你需要安装 ib\_mthca 内核模块和 libmthca 用户空间驱动。

### 用户-内核通信

用户空间通过/dev/infiniband/uverbsN 字符设备与内核进行慢速路径、资源管理操作的通信。快速路径操作通常是通过直接写入硬件寄存器 mmap() 到用户空间来完成的，没有系统调用或上下文切换到内核。

命令是通过在这些设备文件上的 write()s 发送给内核的。ABI 在 drivers/infiniband/include/ib\_user\_verbs.h 中定义。需要内核响应的命令的结构包含一个 64 位字段，用来传递一个指向输出缓冲区的指针。状态作为 write() 系统调用的返回值被返回到用户空间。

## 资源管理

由于所有 IB 资源的创建和销毁都是通过文件描述符传递的命令完成的，所以内核可以跟踪那些被附加到给定用户空间上下文的资源。`ib_uverbs` 模块维护着 `idr` 表，用来在内核指针和不透明的用户空间句柄之间进行转换，这样内核指针就不会暴露给用户空间，而用户空间也无法欺骗内核去跟踪一个假的指针。

这也允许内核在一个进程退出时进行清理，并防止一个进程触及另一个进程的资源。

## 内存固定

直接的用户空间 I/O 要求与作为潜在 I/O 目标的内存区域保持在同一物理地址上。`ib_uverbs` 模块通过 `get_user_pages()` 和 `put_page()` 调用来管理内存区域的固定和解除固定。它还核算进程的 `pinned_vm` 中被固定的内存量，并检查非特权进程是否超过其 `RLIMIT_MEMLOCK` 限制。被多次固定的页面在每次被固定时都会被计数，所以 `pinned_vm` 的值可能会高估一个进程所固定的页面数量。

## /dev 文件

要想用 udev 自动创建适当的字符设备文件，可以采用如下规则：

```
KERNEL=="uverbs*", NAME="infiniband/%k"
```

可以使用。这将创建设备节点，名为：

```
/dev/infiniband/uverbs0
```

等等。由于 InfiniBand 的用户空间 verbs 对于非特权进程来说应该是安全的，因此在 udev 规则中加入适当的 MODE 或 GROUP 可能是有用的。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/power/index.rst

翻译 唐艺舟 Tang Yizhou <[tangyeechou@gmail.com](mailto:tangyeechou@gmail.com)>

\* 电源管理

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题, 请联系该文件的译者, 或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/power/energy-model.rst

翻译 唐艺舟 Tang Yizhou <[tangyeechou@gmail.com](mailto:tangyeechou@gmail.com)>

## 设备能量模型

## 1. 概述

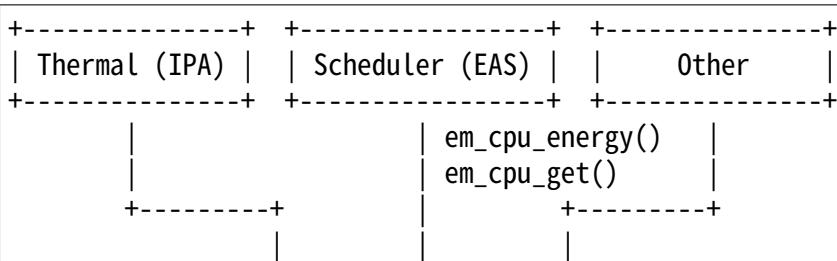
能量模型 (EM) 框架是一种驱动程序与内核子系统之间的接口。其中驱动程序了解不同性能层级的设备所消耗的功率，而内核子系统愿意使用该信息做出能量感知决策。

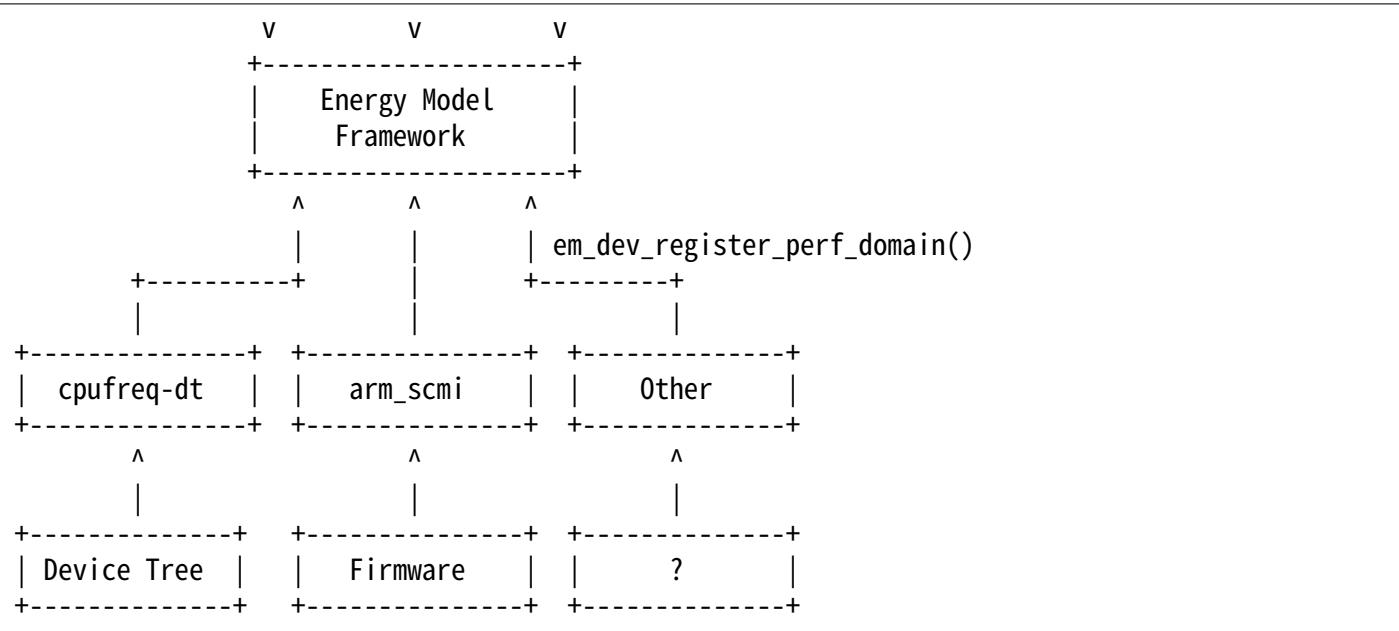
设备所消耗的功率的信息来源在不同的平台上可能有很大的不同。这些功率成本在某些情况下可以使用设备树数据来估算。在其它情况下，固件会更清楚。或者，用户空间可能是最清楚的。以此类推。为了避免每一个客户端子系统对每一种可能的信息源自己重新实现支持，EM 框架作为一个抽象层介入，它在内核中对功率成本表的格式进行标准化，因此能够避免多余的工作。

功率值可以用毫瓦或“抽象刻度”表示。多个子系统可能使用 EM，由系统集成商来检查功率值刻度类型的要求是否满足。可以在能量感知调度器的文档中找到一个例子 [Documentation/scheduler/sched-energy.rst](#)。对于一些子系统，比如热能或 powercap，用“抽象刻度”描述功率值可能会导致问题。这些子系统对过去使用的功率的估算值更感兴趣，因此可能需要真实的毫瓦。这些要求的一个例子可以在智能功率分配 [Documentation/driver-api/thermal/power\\_allocator.rst](#) 文档中找到。

内核子系统可能（基于 EM 内部标志位）实现了对 EM 注册设备是否具有不一致刻度的自动检查。要记住的重要事情是，当功率值以“抽象刻度”表示时，从中推导以毫焦耳为单位的真实能量消耗是不可能的。

下图描述了一个驱动的例子（这里是针对 Arm 的，但该方法适用于任何体系结构），它向 EM 框架提供了功率成本，感兴趣的客户端可从中读取数据：





对于 CPU 设备，EM 框架管理着系统中每个“性能域”的功率成本表。一个性能域是一组性能一起伸缩的 CPU。性能域通常与 CPUFreq 策略具有 1 对 1 映射。一个性能域中的所有 CPU 要求具有相同的微架构。不同性能域中的 CPU 可以有不同的微架构。

## 2. 核心 API

### 2.1 配置选项

必须使能 CONFIG\_ENERGY\_MODEL 才能使用 EM 框架。

### 2.2 性能域的注册

#### “高级” EM 的注册

“高级” EM 因它允许驱动提供更精确的功率模型而得名。它并不受限于框架中的一些已实现的数学公式（就像“简单” EM 那样）。它可以更好地反映每个性能状态的实际功率测量。因此，在 EM 静态功率（漏电流功率）是重要的情况下，应该首选这种注册方式。

驱动程序应通过以下 API 将性能域注册到 EM 框架中：

```
int em_dev_register_perf_domain(struct device *dev, unsigned int nr_states,
                                struct em_data_callback *cb, cpumask_t *cpus, bool milliwatts);
```

驱动程序必须提供一个回调函数，为每个性能状态返回 < 频率, 功率 > 元组。驱动程序提供的回调函数可以自由地从任何相关位置（DT、固件……）以及以任何被认为是必要的方式获取数据。只有对于 CPU 设备，驱动程序必须使用 cpumask 指定性能域的 CPU。对于 CPU 以外的其他设备，最后一个参数必须被设置为 NULL。

最后一个参数“milliwatts”（毫瓦）设置成正确的值是很重要的，使用 EM 的内核子系统可能会依赖这个标志来检查所有的 EM 设备是否使用相同的刻度。如果有不同的刻度，这些子系统可能决定：返回警告/错误，停止工作或崩溃（panic）。

关于实现这个回调函数的驱动程序的例子，参见第 3 节。或者在第 2.4 节阅读这个 API 的更多文档。

### “简单” EM 的注册

“简单” EM 是用框架的辅助函数 `cpufreq_register_em_with_opp()` 注册的。它实现了一个和以下数学公式紧密相关的功率模型：

$$\text{Power} = C * V^2 * f$$

使用这种方法注册的 EM 可能无法正确反映真实设备的物理特性，例如当静态功率（漏电流功率）很重要时。

## 2.3 访问性能域

有两个 API 函数提供对能量模型的访问。`em_cpu_get()` 以 CPU id 为参数，`em_pd_get()` 以设备指针为参数。使用哪个接口取决于子系统，但对于 CPU 设备来说，这两个函数都返回相同的性能域。

对 CPU 的能量模型感兴趣的子系统可以通过 `em_cpu_get()` API 检索它。在创建性能域时分配一次能量模型表，它保存在内存中不被修改。

一个性能域所消耗的能量可以使用 `em_cpu_energy()` API 来估算。该估算假定 CPU 设备使用的 CPUfreq 监管器是 `schedutil`。当前该计算不能提供给其它类型的设备。

关于上述 API 的更多细节可以在 `<linux/energy_model.h>` 或第 2.4 节中找到。

## 2.4 API 的细节描述

参见 `include/linux/energy_model.h` 和 `kernel/power/energy_model.c` 的 kernel doc。

## 3. 驱动示例

CPUFreq 框架支持专用的回调函数，用于为指定的 CPU（们）注册 EM：`cpufreq_driver::register_em()`。这个回调必须为每个特定的驱动程序正确实现，因为框架会在设置过程中适时地调用它。本节提供了一个简单的例子，展示 CPUFreq 驱动在能量模型框架中使用（假的）“foo”协议注册性能域。该驱动实现了一个 `est_power()` 函数提供给 EM 框架：

```
-> drivers/cpufreq/foo_cpufreq.c
01 static int est_power(unsigned long *mW, unsigned long *kHz,
02                      struct device *dev)
03 {
04     long freq, power;
```

```

05     /* 使用 “foo” 协议设置频率上限 */
06     freq = foo_get_freq_ceil(dev, *KHz);
07     if (freq < 0);
08         return freq;
09
10     /* 估算相关频率下设备的功率成本 */
11     power = foo_estimate_power(dev, freq);
12     if (power < 0);
13         return power;
14
15     /* 将这些值返回给 EM 框架 */
16     *mW = power;
17     *KHz = freq;
18
19
20     return 0;
21 }
22
23 static void foo_cpufreq_register_em(struct cpufreq_policy *policy)
24 {
25     struct em_data_callback em_cb = EM_DATA_CB(est_power);
26     struct device *cpu_dev;
27     int nr_opp;
28
29     cpu_dev = get_cpu_device(cpumask_first(policy->cpus));
30
31     /* 查找该策略支持的 OPP 数量 */
32     nr_opp = foo_get_nr_opp(policy);
33
34     /* 并注册新的性能域 */
35     em_dev_register_perf_domain(cpu_dev, nr_opp, &em_cb, policy->cpus,
36                                 true);
37 }
38
39 static struct cpufreq_driver foo_cpufreq_driver = {
40     .register_em = foo_cpufreq_register_em,
41 };

```

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

Original Documentation/power/opp.rst

翻译 唐艺舟 Tang Yizhou <tangyeechou@gmail.com>

## 操作性能值 (OPP) 库

(C) 2009-2010 Nishanth Menon <nm@ti.com>, 德州仪器公司

### 1. 简介

#### 1.1 何为操作性能值 (OPP)?

当今复杂的单片系统 (SoC) 由多个子模块组成，这些子模块会联合工作。在一个执行不同用例的操作系统中，并不是 SoC 中的所有模块都需要一直以最高频率工作。为了促成这一点，SoC 中的子模块被分组为不同域，允许一些域以较低的电压和频率运行，而其它域则以较高的“电压/ 频率对”运行。

设备按域支持的由频率电压对组成的离散的元组的集合，被称为操作性能值 (组)，或 OPPs。

举例来说：

让我们考虑一个支持下述频率、电压值的内存保护单元 (MPU) 设备：{300MHz, 最低电压为 1V}, {800MHz, 最低电压为 1.2V}, {1GHz, 最低电压为 1.3V}

我们能将它们表示为 3 个 OPP，如下述 {Hz, uV} 元组（译注：频率的单位是赫兹，电压的单位是微伏）。

- {3000000000, 1000000}
- {8000000000, 1200000}
- {10000000000, 1300000}

#### 1.2 操作性能值库

OPP 库提供了一组辅助函数来组织和查询 OPP 信息。该库位于 drivers/opp/ 目录下，其头文件位于 include/linux/pm\_opp.h 中。OPP 库可以通过开启 CONFIG\_PM\_OPP 来启用。某些 SoC，如德州仪器的 OMAP 框架允许在不需要 cpufreq 的情况下可选地在某一 OPP 下启动。

OPP 库的典型用法如下：



OPP 层期望每个域由一个唯一的设备指针来表示。SoC 框架在 OPP 层为每个设备注册了一组初始 OPP。这个链表的长度被期望是一个最优化的小数字，通常每个设备大约 5 个。初始链表包含了一个 OPP 集合，这个集合被期望能在系统中安全使能。

## 关于 OPP 可用性的说明

随着系统的运行，SoC 框架可能会基于各种外部因素选择让某些 OPP 在每个设备上可用或不可用，示例：温度管理或其它异常场景中，SoC 框架可能会选择禁用一个较高频率的 OPP 以安全地继续运行，直到该 OPP 被重新启用（如果可能）。

OPP 库在它的实现中达成了这个概念。以下操作函数只能对可用的 OPP 使用：`dev_pm_opp_find_freq_{ceil, floor}`, `dev_pm_opp_get_voltage`, `dev_pm_opp_get_freq`, `dev_pm_opp_get_opp_count`。

`dev_pm_opp_find_freq_exact`是用来查找 OPP 指针的，该指针可被用在 `dev_pm_opp_enable/disable` 函数，使一个 OPP 在被需要时变为可用。

警告：如果对一个设备调用 `dev_pm_opp_enable/disable` 函数，OPP 库的用户应该使用 `dev_pm_opp_get_opp_count` 来刷新 OPP 的可用性计数。触发这些的具体机制，或者对有依赖的子系统（比如 cpufreq）的通知机制，都是由使用 OPP 库的 SoC 特定框架酌情处理的。在这些操作中，同样需要注意刷新 cpufreq 表。

## 2. OPP 链表初始注册

SoC 的实现会迭代调用 `dev_pm_opp_add` 函数来增加每个设备的 OPP。预期 SoC 框架将以最优的方式注册 OPP 条目 - 典型的数字范围小于 5。通过注册 OPP 生成的 OPP 链表，在整个设备运行过程中由 OPP 库维护。SoC 框架随后可以使用 `dev_pm_opp_enable / disable` 函数动态地控制 OPP 的可用性。

**dev\_pm\_opp\_add** 为设备指针所指向的特定域添加一个新的 OPP。OPP 是用频率和电压定义的。一旦完成添加，OPP 被认为是可用的，可以用 `dev_pm_opp_enable/disable` 函数来控制其可用性。OPP 库内部用 `dev_pm_opp` 结构体存储并管理这些信息。这个函数可以被 SoC 框架根据 SoC 的使用环境的需求来定义一个最优链表。

警告：不要在中断上下文使用这个函数。

示例：

```
soc_pm_init()
{
    /* 做一些事情 */
    r = dev_pm_opp_add(mpu_dev, 1000000, 900000);
    if (!r) {
        pr_err("%s: unable to register mpu opp(%d)\n", r);
        goto no_cpufreq;
    }
    /* 做一些和 cpufreq 相关的事情 */
no_cpufreq:
    /* 做剩余的事情 */
}
```

### 3. OPP 搜索函数

cpufreq 等高层框架对频率进行操作，为了将频率映射到相应的 OPP，OPP 库提供了便利的函数来搜索 OPP 库内部管理的 OPP 链表。这些搜索函数如果找到匹配的 OPP，将返回指向该 OPP 的指针，否则返回错误。这些错误预计由标准的错误检查，如 IS\_ERR() 来处理，并由调用者采取适当的行动。

这些函数的调用者应在使用完 OPP 后调用 dev\_pm\_opp\_put()。否则，OPP 的内存将永远不会被释放，并导致内存泄露。

**dev\_pm\_opp\_find\_freq\_exact** 根据 精确的频率和可用性来搜索 OPP。这个函数对默认不可用的 OPP 特别有用。例子：在 SoC 框架检测到更高频率可用的情况下，它可以使这个函数在调用 dev\_pm\_opp\_enable 之前找到 OPP：

```
opp = dev_pm_opp_find_freq_exact(dev, 1000000000, false);
dev_pm_opp_put(opp);
/* 不要操作指针.. 只是做有效性检查.. */
if (IS_ERR(opp)) {
    pr_err("frequency not disabled!\n");
    /* 触发合适的操作.. */
} else {
    dev_pm_opp_enable(dev,1000000000);
}
```

注意：这是唯一一个可以搜索不可用 OPP 的函数。

**dev\_pm\_opp\_find\_freq\_floor** 搜索一个 最多提供指定频率的可用 OPP。这个函数在搜索较小的匹配或按频率递减的顺序操作 OPP 信息时很有用。例子：要找的一个设备的最高 OPP：

```
freq = ULONG_MAX;
opp = dev_pm_opp_find_freq_floor(dev, &freq);
dev_pm_opp_put(opp);
```

**dev\_pm\_opp\_find\_freq\_ceil** 搜索一个 最少提供指定频率的可用 OPP。这个函数在搜索较大的匹配或按频率递增的顺序操作 OPP 信息时很有用。例 1：找到一个设备最小的 OPP：

```
freq = 0;
opp = dev_pm_opp_find_freq_ceil(dev, &freq);
dev_pm_opp_put(opp);
```

例：一个 SoC 的 cpufreq\_driver->target 的简易实现：

```
soc_cpufreq_target(..)
{
    /* 做策略检查等操作 */
    /* 找到和请求最接近的频率 */
    opp = dev_pm_opp_find_freq_ceil(dev, &freq);
    dev_pm_opp_put(opp);
    if (!IS_ERR(opp))
        soc_switch_to_freq_voltage(freq);
```

```

    else
        /* 当不能满足请求时，要做的事 */
        /* 做其它事 */
}

```

## 4. OPP 可用性控制函数

在 OPP 库中注册的默认 OPP 链表也许无法满足所有可能的场景。OPP 库提供了一套函数来修改 OPP 链表中的某个 OPP 的可用性。这使得 SoC 框架能够精细地动态控制哪一组 OPP 是可用于操作的。设计这些函数的目的是在诸如考虑温度时 暂时地删除某个 OPP（例如，在温度下降之前不要使用某 OPP）。

**警告：** 不要在中断上下文使用这些函数。

**dev\_pm\_opp\_enable** 使一个 OPP 可用于操作。例子：假设 1GHz 的 OPP 只有在 SoC 温度低于某个阈值时才可用。SoC 框架的实现可能会选择做以下事情：

```

if (cur_temp < temp_low_thresh) {
    /* 若 1GHz 未使能，则使能 */
    opp = dev_pm_opp_find_freq_exact(dev, 1000000000, false);
    dev_pm_opp_put(opp);
    /* 仅仅是错误检查 */
    if (!IS_ERR(opp))
        ret = dev_pm_opp_enable(dev, 1000000000);
    else
        goto try_something_else;
}

```

**dev\_pm\_opp\_disable** 使一个 OPP 不可用于操作。例子：假设 1GHz 的 OPP 只有在 SoC 温度高于某个阈值时才可用。SoC 框架的实现可能会选择做以下事情：

```

if (cur_temp > temp_high_thresh) {
    /* 若 1GHz 已使能，则关闭 */
    opp = dev_pm_opp_find_freq_exact(dev, 1000000000, true);
    dev_pm_opp_put(opp);
    /* 仅仅是错误检查 */
    if (!IS_ERR(opp))
        ret = dev_pm_opp_disable(dev, 1000000000);
    else
        goto try_something_else;
}

```

## 5. OPP 数据检索函数

由于 OPP 库对 OPP 信息进行了抽象化处理，因此需要一组函数来从 dev\_pm\_opp 结构体中提取信息。一旦使用搜索函数检索到一个 OPP 指针，以下函数就可以被 SoC 框架用来检索 OPP 层内部描述的信息。

**dev\_pm\_opp\_get\_voltage** 检索 OPP 指针描述的电压。例子：当 cpufreq 切换到到不同频率时，SoC 框架需要用稳压器框架将 OPP 描述的电压设置到提供电压的电源管理芯片中：

```
soc_switch_to_freq_voltage(freq)
{
    /* 做一些事情 */
    opp = dev_pm_opp_find_freq_ceil(dev, &freq);
    v = dev_pm_opp_get_voltage(opp);
    dev_pm_opp_put(opp);
    if (v)
        regulator_set_voltage(.., v);
    /* 做其它事 */
}
```

**dev\_pm\_opp\_get\_freq** 检索 OPP 指针描述的频率。例子：比方说，SoC 框架使用了几个辅助函数，通过这些函数，我们可以将 OPP 指针传入，而不是传入额外的参数，用来处理一系列数据参数：

```
soc_cpufreq_target(..)
{
    /* 做一些事情.. */
    max_freq = ULONG_MAX;
    max_opp = dev_pm_opp_find_freq_floor(dev,&max_freq);
    requested_opp = dev_pm_opp_find_freq_ceil(dev,&freq);
    if (!IS_ERR(max_opp) && !IS_ERR(requested_opp))
        r = soc_test_validity(max_opp, requested_opp);
    dev_pm_opp_put(max_opp);
    dev_pm_opp_put(requested_opp);
    /* 做其它事 */
}
soc_test_validity(..)
{
    if(dev_pm_opp_get_voltage(max_opp) < dev_pm_opp_get_voltage(requested_opp))
        return -EINVAL;
    if(dev_pm_opp_get_freq(max_opp) < dev_pm_opp_get_freq(requested_opp))
        return -EINVAL;
    /* 做一些事情.. */
}
```

**dev\_pm\_opp\_get\_opp\_count** 检索某个设备可用的 OPP 数量。例子：假设 SoC 中的一个协处理器需要知道某个表中的可用频率，主处理器可以按如下方式发出通知：

```
soc_notify_coproc_available_frequencies()
{
    /* 做一些事情 */
    num_available = dev_pm_opp_get_opp_count(dev);
```

```

speeds = kzalloc(sizeof(u32) * num_available, GFP_KERNEL);
/* 按升序填充表 */
freq = 0;
while (!IS_ERR(opp = dev_pm_opp_find_freq_ceil(dev, &freq))) {
    speeds[i] = freq;
    freq++;
    i++;
    dev_pm_opp_put(opp);
}

soc_notify_coproc(AVAILABLE_FREQS, speeds, num_available);
/* 做其它事 */
}

```

## 6. 数据结构

通常，一个 SoC 包含多个可变电压域。每个域由一个设备指针描述。和 OPP 之间的关系可以按以下方式描述：

```

SoC
|- device 1
|   |- opp 1 (availability, freq, voltage)
|   |- opp 2 ...
...
`- opp n ...
|- device 2
...
`- device m

```

OPP 库维护着一个内部链表，SoC 框架使用上文描述的各个函数来填充和访问。然而，描述真实 OPP 和域的结构体是 OPP 库自身的内部组成，以允许合适的抽象在不同系统中得到复用。

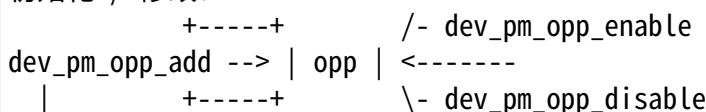
**struct dev\_pm\_opp** OPP 库的内部数据结构，用于表示一个 OPP。除了频率、电压、可用性信息外，它还包含 OPP 库运行所需的内部统计信息。指向这个结构体的指针被提供给用户（比如 SoC 框架）使用，在与 OPP 层的交互中作为 OPP 的标识符。

**警告：** 结构体 `dev_pm_opp` 的指针不应该由用户解析或修改。一个实例的默认值由 `dev_pm_opp_add` 填充，但 OPP 的可用性由 `dev_pm_opp_enable/disable` 函数修改。

**struct device** 这用于向 OPP 层标识一个域。设备的性质和它的实现是由 OPP 库的用户决定的，如 SoC 框架。

总体来说，以一个简化的视角看，对数据结构的操作可以描述为下面各图：

初始化 / 修改：



```
\-----> domain_info(device)
```

搜索函数:

```
    /-- dev_pm_opp_find_freq_ceil ---\ +----+
domain_info<--- dev_pm_opp_find_freq_exact -----> | opp |
    \-- dev_pm_opp_find_freq_floor ---/ +----+
```

检索函数:

```
+----+     /- dev_pm_opp_get_voltage
| opp | <---
+----+     \- dev_pm_opp_get_freq
```

```
domain_info <- dev_pm_opp_get_opp_count
```

TODOList:

- apm-acpi
- basic-pm-debugging
- charger-manager
- drivers-testing
- freezing-of-tasks
- pci
- pm\_qos\_interface
- power\_supply\_class
- runtime\_pm
- s2ram
- suspend-and-cpuhotplug
- suspend-and-interrupts
- swsusp-and-swap-files
- swsusp-dmcrypt
- swsusp
- video
- tricks
- userland-swsusp
- powercap/powercap
- powercap/dtpm

- regulator/consumer
- regulator/design
- regulator/machine
- regulator/overview
- regulator/regulator

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

**Original** Documentation/virt/index.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 时奎亮 Alex Shi <[alexs@kernel.org](mailto:alexs@kernel.org)>

## \* Linux 虚拟化支持

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

**Original** Documentation/virt/paravirt\_ops.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 陈飞杨 Feiyang Chen <[chenfeiyang@loongson.cn](mailto:chenfeiyang@loongson.cn)> 时奎亮 Alex Shi <[alexs@kernel.org](mailto:alexs@kernel.org)>

### 半虚拟化操作

Linux 提供了对不同管理程序虚拟化技术的支持。历史上，为了支持不同的虚拟机超级管理器（hypervisor，下文简称超级管理器），需要不同的二进制内核，这个限制已经被 `pv_ops` 移除了。Linux `pv_ops` 是一个虚拟化 API，它能够支持不同的管理程序。它允许每个管理程序优先于关键操作，并允许单一的内核二进制文件在所有支持的执行环境中运行，包括本机——没有任何管理程序。

`pv_ops` 提供了一组函数指针，代表了与低级关键指令和各领域高级功能相对应的操作。`pv-ops` 允许在运行时进行优化，在启动时对低级关键操作进行二进制修补。

`pv_ops` 操作被分为三类：

- **简单的间接调用** 这些操作对应于高水平的函数，众所周知，间接调用的开销并不十分重要。
- **间接调用，允许用二进制补丁进行优化** 通常情况下，这些操作对应于低级别的关键指令。它们被频繁地调用，并且是对性能关键。开销是非常重要的。
- **一套用于手写汇编代码的宏程序** 手写的汇编代码 (.S 文件) 也需要半虚拟化，因为它们包括敏感指令或其中的一些代码路径对性能非常关键。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:<alexs@kernel.org>)。

---

**Original** Documentation/virt/guest-halt-polling.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 时奎亮 Alex Shi <[alexs@kernel.org](mailto:alexs@kernel.org)>

### 客户机停机轮询机制（Guest halt polling）

`cpuidle_haltpoll` 驱动，与 `haltpoll` 管理器一起，允许客户机 vcpus 在停机前轮询一定的时间。

这为物理机侧的轮询提供了以下好处：

- 1) 在执行轮询时，POLL 标志被设置，这允许远程 vCPU 在执行唤醒时避免发送 IPI（以及处理 IPI 的相关成本）。
- 2) 可以避免虚拟机退出的成本。

客户机侧轮询的缺点是，即使在物理机中的其他可运行任务中也会进行轮询。

其基本逻辑如下。一个全局值，即 `guest_halt_poll_ns`，是由用户配置的，表示允许轮询的最大时间量。这个值是固定的。

每个 vcpu 都有一个可调整的 `guest_halt_poll_ns` (" per-cpu guest\_halt\_poll\_ns"), 它由算法响应事件进行调整 (解释如下)。

## 模块参数

`haltpoll` 管理器有 5 个可调整的模块参数:

1) `guest_halt_poll_ns`:

轮询停机前执行的最大时间, 以纳秒为单位。

默认值: 200000

2) `guest_halt_poll_shrink`:

当唤醒事件发生在全局的 `guest_halt_poll_ns` 之后, 用于缩减每个 CPU 的 `guest_halt_poll_ns` 的划分系数。

默认值: 2

3) `guest_halt_poll_grow`:

当事件发生在 per-cpu `guest_halt_poll_ns` 之后但在 global `guest_halt_poll_ns` 之前, 用于增长 per-cpu `guest_halt_poll_ns` 的乘法系数。

默认值: 2

4) `guest_halt_poll_grow_start`:

在系统空闲的情况下, 每个 cpu `guest_halt_poll_ns` 最终达到零。这个值设置了增长时的初始每 cpu `guest_halt_poll_ns`。这个值可以从 10000 开始增加, 以避免在最初的增长阶段出现失误。:

10k, 20k, 40k, …(例如, 假设 `guest_halt_poll_grow=2`).

默认值: 50000

5) `guest_halt_poll_allow_shrink`:

允许缩减的 Bool 参数。设置为 N 以避免它 (一旦达到全局的 `guest_halt_poll_ns` 值, 每 CPU 的 `guest_halt_poll_ns` 将保持高位)。

默认值: Y

模块参数可以从 Debugfs 文件中设置, 在:

```
/sys/module/haltpoll/parameters/
```

### 进一步说明

- 在设置 guest\_halt\_poll\_ns 参数时应该小心，因为一个大的值有可能使几乎是完全空闲机器上的 cpu 使用率达到 100%。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/virt/ne\_overview.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 时奎亮 Alex Shi <[alexs@kernel.org](mailto:alexs@kernel.org)>

## Nitro Enclaves

### 概述

Nitro Enclaves (NE) 是亚马逊弹性计算云 (EC2) 的一项新功能，允许客户在 EC2 实例中划分出孤立的计算环境 [1]。

例如，一个处理敏感数据并在虚拟机中运行的应用程序，可以与在同一虚拟机中运行的其他应用程序分开。然后，这个应用程序在一个独立于主虚拟机的虚拟机中运行，即 enclave。

一个 enclave 与催生它的虚拟机一起运行。这种设置符合低延迟应用的需要。为 enclave 分配的资源，如内存和 CPU，是从主虚拟机中分割出来的。每个 enclave 都被映射到一个运行在主虚拟机中的进程，该进程通过一个 ioctl 接口与 NE 驱动进行通信。

在这个意义上，有两个组成部分。

1. 一个 enclave 抽象进程——一个运行在主虚拟机客体中的用户空间进程，它使用 NE 驱动提供的 ioctl 接口来生成一个 enclave 虚拟机（这就是下面的 2）。

有一个 NE 模拟的 PCI 设备暴露给主虚拟机。这个新的 PCI 设备的驱动被包含在 NE 驱动中。

ioctl 逻辑被映射到 PCI 设备命令，例如，NE\_START\_ENCLAVE ioctl 映射到一个 enclave 启动 PCI 命令。然后，PCI 设备命令被翻译成在管理程序方面采取的行动；也就是在运行主虚拟机的主机上运行的 Nitro 管理程序。Nitro 管理程序是基于 KVM 核心技术的。

2. enclave 本身——一个运行在与催生它的主虚拟机相同的主机上的虚拟机。内存和 CPU 从主虚拟机中分割出来，专门用于 enclave 虚拟机。enclave 没有连接持久性存储。

从主虚拟机中分割出来并给 enclave 的内存区域需要对齐 2 MiB/1 GiB 物理连续的内存区域（或这个大小的倍数，如 8 MiB）。该内存可以通过使用 hugetlbfs 从用户空间分配 [2][3]。一个 enclave 的内存大小需要至少 64 MiB。enclave 内存和 CPU 需要来自同一个 NUMA 节点。

一个 enclave 在专用的核心上运行。CPU 0 及其同级别的 CPU 需要保持对主虚拟机的可用性。CPU 池必须由具有管理能力的用户为 NE 目的进行设置。关于 CPU 池的格式，请看内核文档 [4] 中的 cpu list 部分。

enclave 通过本地通信通道与主虚拟机进行通信，使用 virtio-vsock[5]。主虚拟机有 virtio-pci vsock 模拟设备，而飞地虚拟机有 virtio-mmio vsock 模拟设备。vsock 设备使用 eventfd 作为信令。enclave 虚拟机看到通常的接口——本地 APIC 和 IOAPIC——从 virtio-vsock 设备获得中断。virtio-mmio 设备被放置在典型的 4 GiB 以下的内存中。

在 enclave 中运行的应用程序需要和将在 enclave 虚拟机中运行的操作系统（如内核、ramdisk、init）一起被打包到 enclave 镜像中。enclave 虚拟机有自己的内核并遵循标准的 Linux 启动协议 [6]。

内核 bzImage、内核命令行、ramdisk (s) 是 enclave 镜像格式 (EIF) 的一部分；另外还有一个 EIF 头，包括元数据，如 magic number、eif 版本、镜像大小和 CRC。

哈希值是为整个 enclave 镜像 (EIF)、内核和 ramdisk (s) 计算的。例如，这被用来检查在 enclave 虚拟机中加载的 enclave 镜像是否是打算运行的那个。

这些加密测量包括在由 Nitro 超级管理器成的签名证明文件中，并进一步用来证明 enclave 的身份；KMS 是 NE 集成的服务的一个例子，它检查证明文件。

enclave 镜像 (EIF) 被加载到 enclave 内存中，偏移量为 8 MiB。enclave 中的初始进程连接到主虚拟机的 vsock CID 和一个预定义的端口-9000，以发送一个心跳值-0xb7。这个机制用于在主虚拟机中检查 enclave 是否已经启动。主虚拟机的 CID 是 3。

如果 enclave 虚拟机崩溃或优雅地退出，NE 驱动会收到一个中断事件。这个事件会通过轮询通知机制进一步发送到运行在主虚拟机中的用户空间 enclave 进程。然后，用户空间 enclave 进程就可以退出了。

[1] <https://aws.amazon.com/ec2/nitro/nitro-enclaves/> [2] <https://www.kernel.org/doc/html/latest/admin-guide/mm/hugetlbpage.html> [3] <https://lwn.net/Articles/807108/> [4] <https://www.kernel.org/doc/html/latest/admin-guide/kernel-parameters.html> [5] <https://man7.org/linux/man-pages/man7/vsock.7.html> [6] <https://www.kernel.org/doc/html/latest/x86/boot.html>

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

**Original** Documentation/virt/acrn/index.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 时奎亮 Alex Shi <[alexs@kernel.org](mailto:alexs@kernel.org)>

## ACRN 超级管理器

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

Original Documentation/virt/acrn/introduction.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

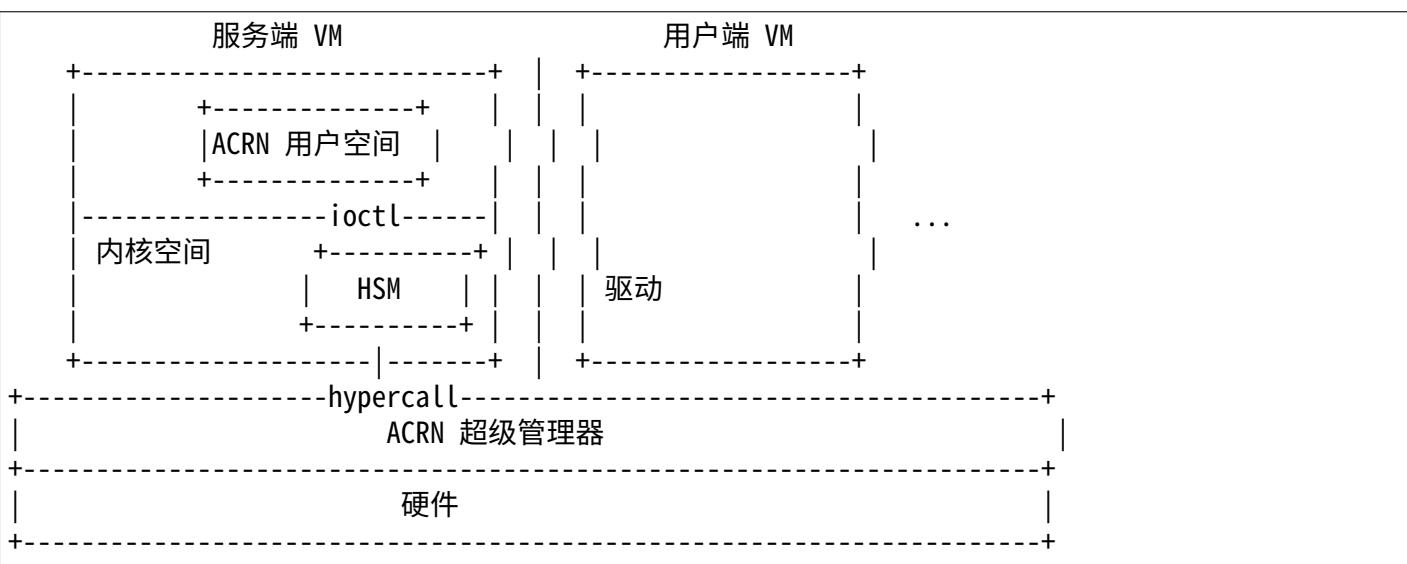
校译 时奎亮 Alex Shi <[alexs@kernel.org](mailto:alexs@kernel.org)>

## ACRN 超级管理器介绍

ACRN 超级管理器是一个第一类超级管理器，直接在裸机硬件上运行。它有一个特权管理虚拟机，称为服务虚拟机，用于管理用户虚拟机和进行 I/O 仿真。

ACRN 用户空间是一个运行在服务虚拟机中的应用程序，它根据命令行配置为用户虚拟机仿真设备。ACRN 管理程序服务模块（HSM）是服务虚拟机中的一个内核模块，为 ACRN 用户空间提供管理程序服务。

下图展示了该架构。



ACRN 用户空间为用户虚拟机分配内存，配置和初始化用户虚拟机使用的设备，加载虚拟引导程序，初始化虚拟 CPU 状态，处理来自用户虚拟机的 I/O 请求访问。它使用 ioctls 来与 HSM 通信。HSM 通过与

ACRN 超级管理器的 hypercalls 进行交互来实现管理服务。HSM 向用户空间输出一个 char 设备接口 (`/dev/acrn_hsm`)。

ACRN 超级管理器是开源的，任何人都可以贡献。源码库在 <https://github.com/projectacrn/acrn-hypervisor>。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:<alexs@kernel.org>)。

**Original** Documentation/virt/acrn/io-request.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 时奎亮 Alex Shi <[alexs@kernel.org](mailto:alexs@kernel.org)>

## I/O 请求处理

客户虚拟机的 I/O 请求由超级管理器构建，由 ACRN 超级管理器服务模块分发到与 I/O 请求的地址范围相对应的 I/O 客户端。I/O 请求处理的细节将在以下章节描述。

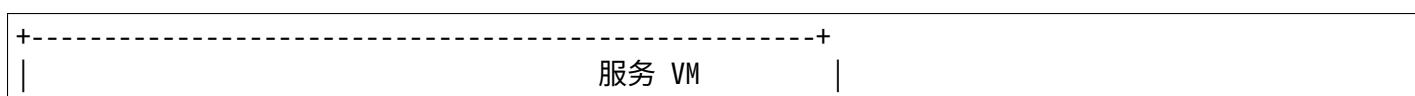
### 1. I/O 请求

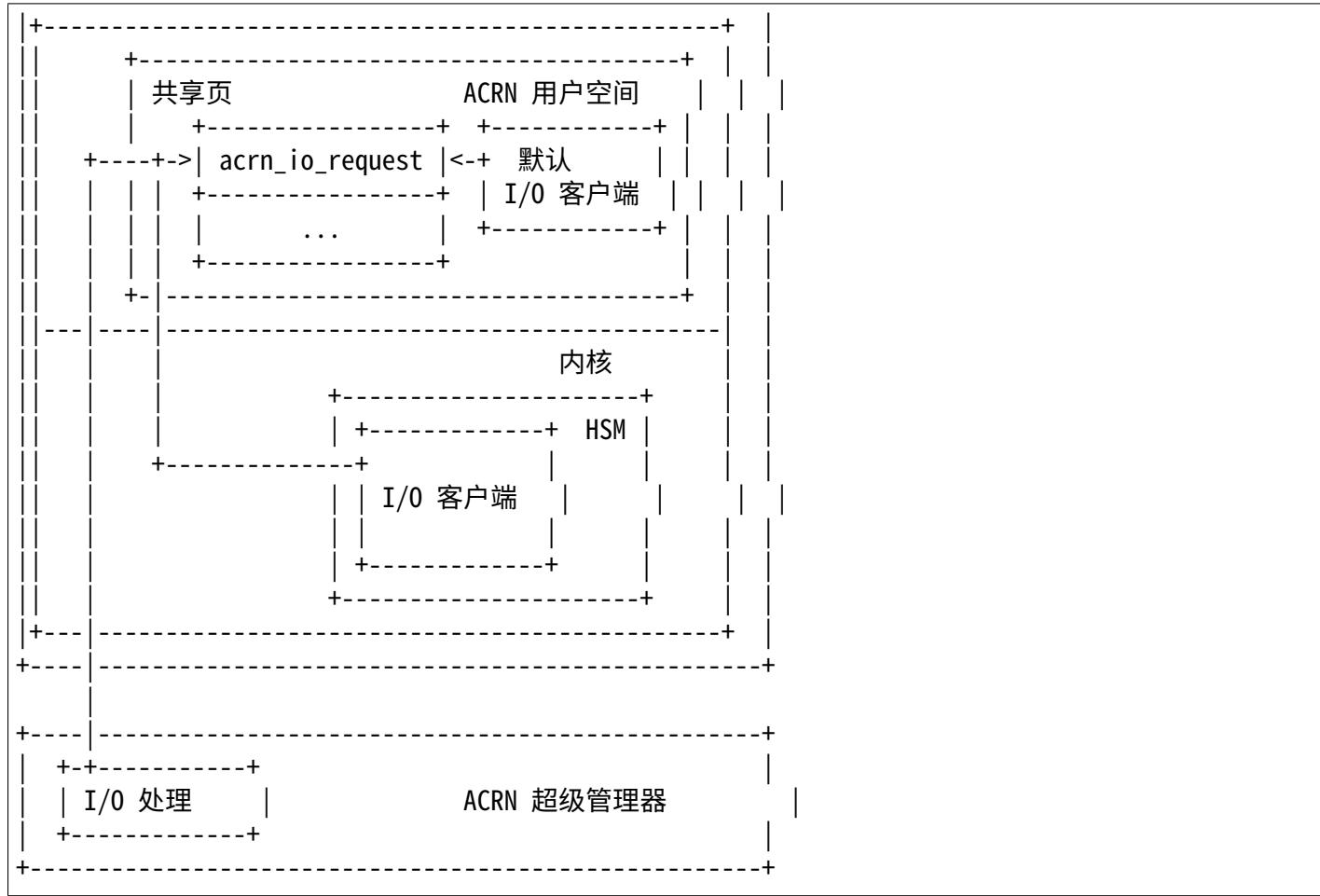
对于每个客户虚拟机，有一个共享的 4KB 字节的内存区域，用于超级管理器和服务虚拟机之间的 I/O 请求通信。一个 I/O 请求是一个 256 字节的结构体缓冲区，它是“`acrn_io_request`”结构体，当客户虚拟机中发生被困的 I/O 访问时，由超级管理器的 I/O 处理器填充。服务虚拟机中的 ACRN 用户空间首先分配一个 4KB 字节的页面，并将缓冲区的 GPA（客户物理地址）传递给管理平台。缓冲区被用作 16 个 I/O 请求槽的数组，每个 I/O 请求槽为 256 字节。这个数组是按 vCPU ID 索引的。

### 2. I/O 客户端

一个 I/O 客户端负责处理客户虚拟机的 I/O 请求，其访问的 GPA 在一定范围内。每个客户虚拟机可以关联多个 I/O 客户端。每个客户虚拟机都有一个特殊的客户端，称为默认客户端，负责处理所有不在其他客户端范围内的 I/O 请求。ACRN 用户空间充当每个客户虚拟机的默认客户端。

下面的图示显示了 I/O 请求共享缓冲区、I/O 请求和 I/O 客户端之间的关系。





### 3. I/O 请求状态转换

一个 ACRN I/O 请求的状态转换如下。

FREE -> PENDING -> PROCESSING -> COMPLETE -> FREE -> ...

- FREE: 这个 I/O 请求槽是空的
- PENDING: 在这个槽位上有一个有效的 I/O 请求正在等待
- PROCESSING: 正在处理 I/O 请求
- COMPLETE: 该 I/O 请求已被处理

处于 COMPLETE 或 FREE 状态的 I/O 请求是由超级管理器拥有的。HSM 和 ACRN 用户空间负责处理其他的。

## 4. I/O 请求的处理流程

- a. 当客户虚拟机中发生被陷入的 I/O 访问时，超级管理器的 I/O 处理程序将把 I/O 请求填充为 PENDING 状态。
- b. 超级管理器向服务虚拟机发出一个向上调用，这是一个通知中断。
- c. upcall 处理程序会安排一个工作者来调度 I/O 请求。
- d. 工作者寻找 PENDING I/O 请求，根据 I/O 访问的地址将其分配给不同的注册客户，将其状态更新为 PROCESSING，并通知相应的客户进行处理。
- e. 被通知的客户端处理指定的 I/O 请求。
- f. HSM 将 I/O 请求状态更新为 COMPLETE，并通过 hypercalls 通知超级管理器完成。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

**Original** Documentation/virt/acrn/cpuid.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 时奎亮 Alex Shi <[alexs@kernel.org](mailto:alexs@kernel.org)>

### ACRN CPUID 位域

在 ACRN 超级管理器上运行的客户虚拟机可以使用 CPUID 检查其一些功能。

ACRN 的 cpuid 函数是：

函数: 0x40000000

返回:

```
eax = 0x40000001
ebx = 0x4e524341
ecx = 0x4e524341
edx = 0x4e524341
```

注意，ebx, ecx 和 edx 中的这个值对应于字符串“ACRNACRNACRN”。eax 中的值对应于这个叶子中存在的最大 cpuid 函数，如果将来有更多的函数加入，将被更新。

函数: define ACRN\_CPUID\_FEATURES (0x40000001)

返回:

```
ebx, ecx, edx  
eax = an OR'ed group of (1 << flag)
```

其中 `flag` 的定义如下:

标志	值	描述
ACRN_FEATURE_PRIVILEGED_VM	0	客户虚拟机是一个有特权的虚拟机

函数: 0x40000010

返回:

```
ebx, ecx, edx  
eax = (Virtual) TSC frequency in kHz.
```

TODOLIST:

kvm/index uml/user\_mode\_linux\_howto\_v2

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** ../../sound/index

**Translator** Huang Jianghui <[huangjianghui@uniontech.com](mailto:huangjianghui@uniontech.com)>

### \* Linux 声音子系统文档

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** ../../sound/hd-audio/index

**Translator** Huang Jianghui <[huangjianghui@uniontech.com](mailto:huangjianghui@uniontech.com)>

## 高清音频

**Chinese translator: Huang Jianghui <[huangjianghui@uniontech.com](mailto:huangjianghui@uniontech.com)>**

### orphan

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

## 以下为正文

### 高清音频编解码器特定混音器控件

此文件解释特定于编解码器的混音器控件.

#### 瑞昱编解码器

**声道模式** 这是一个用于更改环绕声道设置的枚举控件，仅在环绕声道打开时显示出现。它给出要使用的通道数：“2ch”，“4ch”，“6ch”，和“8ch”。根据配置，这还控制多 I/O 插孔的插孔重分配。

**自动静音模式** 这是一个枚举控件，用于更改耳机和线路输出插孔的自动静音行为。如果内置扬声器、耳机和/或线路输出插孔在机器上可用，则显示该控件。当只有耳机或者线路输出的时候，它给出“禁用”和“启用”状态。当启用后，插孔插入后扬声器会自动静音。

当耳机和线路输出插孔都存在时，它给出“禁用”、“仅扬声器”和“线路输出 + 扬声器”。当“仅扬声器”被选择，插入耳机或者线路输出插孔可使扬声器静音，但不会使线路输出静音。当线路输出 + 扬声器被选择，插入耳机插孔会同时使扬声器和线路输出静音。

### 矽玛特编解码器

**模拟环回** 此控件启用/禁用模拟环回电路。只有在编解码器提示中将“lookback”设置为真时才会出现 (见 HD-Audio.txt)。请注意，在某些编解码器上，模拟环回和正常 PCM 播放是独占的，即当此选项打开时，您将听不到任何 PCM 流。

**交换中置/低频** 交换中置和低频通道顺序，通常情况下，左侧对应中置，右侧对应低频，启动此项后，左边低频，右边中置。

**耳机作为线路输出** 当此控制开启时，将耳机视为线路输出插孔。也就是说，耳机不会自动静音其他线路输出，没有耳机放大器被设置到引脚上。

**麦克风插口模式、线路插孔模式等** 这些枚举控制输入插孔引脚的方向和偏置。根据插孔类型，它可以设置为“麦克风输入”和“线路输入”以确定输入偏置，或者当引脚是环绕声道的多 I/O 插孔时，它可以设置为“线路输出”。

### 威盛编解码器

**智能 5.1** 一个枚举控件，用于为环绕输出重新分配多个 I/O 插孔的任务。当它打开时，相应的输入插孔（通常是线路输入和麦克风输入）被切换为环绕和中央低频输出插孔。

**独立耳机** 启用此枚举控制时，耳机输出从单个流（第三个 PCM，如 hw:0,2）而不是主流路由。如果耳机 DAC 与侧边或中央低频通道 DAC 共享，则 DAC 将自动切换到耳机。

**环回混合** 一个用于确定是否启动了模拟环回路由的枚举控件。当它启用后，模拟环回路由到前置通道。同样，耳机与扬声器输出也采用相同的路径。作为一个副作用，当设置此模式后，单个音量控制将不再适用于耳机和扬声器，因为只有一个 DAC 连接到混音器小部件。

**动态电源控制** 此控件决定是否启动每个插孔的动态电源控制检测。启用时，根据插孔的插入情况动态更改组件的电源状态 (D0/D3) 以节省电量消耗。但是，如果您的系统没有提供正确的插孔检测，这将无法工作；在这种情况下，请关闭此控件。

**插孔检测** 此控件仅为 VT1708 编解码器提供，它不会为每个插孔插拔提供适当的未请求事件。当此控件打开，驱动将轮询插孔检测，以便耳机自动静音可以工作，而关闭此控件将降低功耗。

### 科胜讯编解码器

**自动静音模式** 见瑞昱解码器

## 模拟编解码器

**通道模式** 这是一个用于更改环绕声道设置的枚举控件，仅在环绕声道可用时显示。它提供了能被使用的通道数：“2ch”、“4ch”和“6ch”。根据配置，这还控制多 I/O 插孔的插孔重分配。

**独立耳机** 启动此枚举控制后，耳机输出从单个流（第三个 PCM，如 hw:0,2）而不是主流路由。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/filesystems/index.rst

**Translator** Wang Wenhui <[wenhu.wang@vivo.com](mailto:wenhu.wang@vivo.com)>

## \* Linux Kernel 中的文件系统

这份正在开发的手册或许在未来某个辉煌的日子里以易懂的形式将 Linux 虚拟文件系统（VFS）层以及基于其上的各种文件系统如何工作呈现给大家。当前可以看到下面的内容。

### 文件系统

文件系统实现文档。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/filesystems/virtiofs.rst

译者

中文版维护者：王文虎 Wang Wenhui <[wenhu.wang@vivo.com](mailto:wenhu.wang@vivo.com)>

中文版翻译者：王文虎 Wang Wenhui <[wenhu.wang@vivo.com](mailto:wenhu.wang@vivo.com)>

中文版校译者：王文虎 Wang Wenhui <[wenhu.wang@vivo.com](mailto:wenhu.wang@vivo.com)>

### virtiofs: virtio-fs 主机 <-> 客机共享文件系统

- Copyright (C) 2020 Vivo Communication Technology Co. Ltd.

#### 介绍

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

客机通常需要访问主机或者远程系统上的文件。使用场景包括：在新客机安装时让文件对其可见；从主机上的根文件系统启动；对无状态或临时客机提供持久存储和在客机之间共享目录。

尽管在某些任务可能通过使用已有的网络文件系统完成，但是却需要非常难以自动化的配置步骤，且将存储网络暴露给客机。而 virtio-fs 设备通过提供不经过网络的文件系统访问文件的设计方式解决了这些问题。

另外，virtio-fs 设备发挥了主客机共存的优点提高了性能，并且提供了网络文件系统所不具备的一些语义功能。

#### 用法

以``myfs``标签将文件系统挂载到``/mnt``：

```
guest# mount -t virtiofs myfs /mnt
```

请查阅 <https://virtio-fs.gitlab.io/> 了解配置 QEMU 和 virtiofsd 守护程序的详细信息。

#### 内幕

由于 virtio-fs 设备将 FUSE 协议用于文件系统请求，因此 Linux 的 virtiofs 文件系统与 FUSE 文件系统客户端紧密集成在一起。客机充当 FUSE 客户端而主机充当 FUSE 服务器，内核与用户空间之间的/dev/fuse 接口由 virtio-fs 设备接口代替。

FUSE 请求被置于虚拟队列中由主机处理。主机填充缓冲区中的响应部分，而客机处理请求的完成部分。

将/dev/fuse 映射到虚拟队列需要解决/dev/fuse 和虚拟队列之间语义上的差异。每次读取/dev/fuse 设备时，FUSE 客户端都可以选择要传输的请求，从而可以使某些请求优先于其他请求。虚拟队列有其队列语义，无法更改已入队请求的顺序。在虚拟队列已满的情况下尤其关键，因为此时不可能加入高优先级的请求。为了解决此差异，virtio-fs 设备采用“hiprio”（高优先级）虚拟队列，专门用于有别于普通请求的高优先级请求。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题, 请联系该文件的译者, 或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

## Original Documentation/filesystems/debugfs.rst

### Debugfs

译者

中文版维护者: 罗楚成 Chucheng Luo <[luochucheng@vivo.com](mailto:luochucheng@vivo.com)>

中文版翻译者: 罗楚成 Chucheng Luo <[luochucheng@vivo.com](mailto:luochucheng@vivo.com)>

中文版校译者: 罗楚成 Chucheng Luo <[luochucheng@vivo.com](mailto:luochucheng@vivo.com)>

版权所有 2020 罗楚成 <[luochucheng@vivo.com](mailto:luochucheng@vivo.com)>

Debugfs 是内核开发人员在用户空间获取信息的简单方法。与/proc 不同, proc 只提供进程信息。也不像 sysfs, 具有严格的“每个文件一个值”的规则。debugfs 根本没有规则, 开发人员可以在这里放置他们想要的任何信息。debugfs 文件系统也不能用作稳定的 ABI 接口。从理论上讲, debugfs 导出文件的时候没有任何约束。但是 [1] 实际情况并不总是那么简单。即使是 debugfs 接口, 也最好根据需要进行设计, 并尽量保持接口不变。

Debugfs 通常使用以下命令安装:

```
mount -t debugfs none /sys/kernel/debug
```

(或等效的/etc/fstab 行)。debugfs 根目录默认仅可由 root 用户访问。要更改对文件树的访问, 请使用“uid”, “gid” 和 “mode” 挂载选项。请注意, debugfs API 仅按照 GPL 协议导出到模块。

使用 debugfs 的代码应包含 <linux/debugfs.h>。然后, 首先是创建至少一个目录来保存一组 debugfs 文件:

```
struct dentry *debugfs_create_dir(const char *name, struct dentry *parent);
```

如果成功, 此调用将在指定的父目录下创建一个名为 name 的目录。如果 parent 参数为空, 则会在 debugfs 根目录中创建。创建目录成功时, 返回值是一个指向 dentry 结构体的指针。该 dentry 结构体的指针可用于在目录中创建文件 (以及最后将其清理干净)。ERR\_PTR (-ERROR) 返回值表明出错。如果返回 ERR\_PTR (-ENODEV), 则表明内核是在没有 debugfs 支持的情况下构建的, 并且下述函数都不会起作用。

在 debugfs 目录中创建文件的最通用方法是:

```
struct dentry *debugfs_create_file(const char *name, umode_t mode,
                                  struct dentry *parent, void *data,
                                  const struct file_operations *fops);
```

在这里, name 是要创建的文件的名称, mode 描述了访问文件应具有的权限, parent 指向应该保存文件的目录, data 将存储在产生的 inode 结构体的 i\_private 字段中, 而 fops 是一组文件操作函数, 这些函数中

实现文件操作的具体行为。至少，`read()` 和/或 `write()` 操作应提供；其他可以根据需要包括在内。同样的，返回值将是指向创建文件的 `dentry` 指针，错误时返回 `ERR_PTR (-ERROR)`，系统不支持 `debugfs` 时返回值为 `ERR_PTR (-ENODEV)`。创建一个初始大小的文件，可以使用以下函数代替：

```
struct dentry *debugfs_create_file_size(const char *name, umode_t mode,
                                         struct dentry *parent, void *data,
                                         const struct file_operations *fops,
                                         loff_t file_size);
```

`file_size` 是初始文件大小。其他参数跟函数 `debugfs_create_file` 的相同。

在许多情况下，没必要自己去创建一组文件操作；对于一些简单的情况，`debugfs` 代码提供了许多帮助函数。包含单个整数值的文件可以使用以下任何一项创建：

```
void debugfs_create_u8(const char *name, umode_t mode,
                      struct dentry *parent, u8 *value);
void debugfs_create_u16(const char *name, umode_t mode,
                      struct dentry *parent, u16 *value);
struct dentry *debugfs_create_u32(const char *name, umode_t mode,
                                 struct dentry *parent, u32 *value);
void debugfs_create_u64(const char *name, umode_t mode,
                      struct dentry *parent, u64 *value);
```

这些文件支持读取和写入给定值。如果某个文件不支持写入，只需根据需要设置 `mode` 参数位。这些文件中的值以十进制表示；如果需要使用十六进制，可以使用以下函数替代：

```
void debugfs_create_x8(const char *name, umode_t mode,
                      struct dentry *parent, u8 *value);
void debugfs_create_x16(const char *name, umode_t mode,
                      struct dentry *parent, u16 *value);
void debugfs_create_x32(const char *name, umode_t mode,
                      struct dentry *parent, u32 *value);
void debugfs_create_x64(const char *name, umode_t mode,
                      struct dentry *parent, u64 *value);
```

这些功能只有在开发人员知道导出值的大小的时候才有用。某些数据类型在不同的架构上有不同的宽度，这样会使情况变得有些复杂。在这种特殊情况下可以使用以下函数：

```
void debugfs_create_size_t(const char *name, umode_t mode,
                           struct dentry *parent, size_t *value);
```

不出所料，此函数将创建一个 `debugfs` 文件来表示类型为 `size_t` 的变量。

同样地，也有导出无符号长整型变量的函数，分别以十进制和十六进制表示如下：

```
struct dentry *debugfs_create_ulong(const char *name, umode_t mode,
                                    struct dentry *parent,
                                    unsigned long *value);
void debugfs_create_xul(const char *name, umode_t mode,
                      struct dentry *parent, unsigned long *value);
```

布尔值可以通过以下方式放置在 debugfs 中:

```
struct dentry *debugfs_create_bool(const char *name, umode_t mode,
                                 struct dentry *parent, bool *value);
```

读取结果文件将产生 Y (对于非零值) 或 N, 后跟换行符写入的时候, 它只接受大写或小写值或 1 或 0。任何其他输入将被忽略。

同样, atomic\_t 类型的值也可以放置在 debugfs 中:

```
void debugfs_create_atomic_t(const char *name, umode_t mode,
                           struct dentry *parent, atomic_t *value)
```

读取此文件将获得 atomic\_t 值, 写入此文件将设置 atomic\_t 值。

另一个选择是通过以下结构体和函数导出一个任意二进制数据块:

```
struct debugfs_blob_wrapper {
    void *data;
    unsigned long size;
};

struct dentry *debugfs_create_blob(const char *name, umode_t mode,
                                 struct dentry *parent,
                                 struct debugfs_blob_wrapper *blob);
```

读取此文件将返回由指针指向 debugfs\_blob\_wrapper 结构体的数据。一些驱动使用 “blobs” 作为一种返回几行 (静态) 格式化文本的简单方法。这个函数可用于导出二进制信息, 但似乎在主线中没有任何代码这样做。请注意, 使用 debugfs\_create\_blob () 命令创建的所有文件是只读的。

如果您要转储一个寄存器块 (在开发过程中经常会这么做, 但是这样的调试代码很少上传到主线中。Debugfs 提供两个函数: 一个用于创建仅寄存器文件, 另一个把一个寄存器块插入一个顺序文件中:

```
struct debugfs_reg32 {
    char *name;
    unsigned long offset;
};

struct debugfs_regset32 {
    struct debugfs_reg32 *regs;
    int nregs;
    void __iomem *base;
};

struct dentry *debugfs_create_regset32(const char *name, umode_t mode,
                                       struct dentry *parent,
                                       struct debugfs_regset32 *regset);

void debugfs_print_regs32(struct seq_file *s, struct debugfs_reg32 *regs,
                          int nregs, void __iomem *base, char *prefix);
```

“base”参数可能为 0，但您可能需要使用 `_stringify` 构建 `reg32` 数组，实际上有许多寄存器名称（宏）是寄存器块在基址上的字节偏移量。

如果要在 `debugfs` 中转储 `u32` 数组，可以使用以下函数创建文件：

```
void debugfs_create_u32_array(const char *name, umode_t mode,
                               struct dentry *parent,
                               u32 *array, u32 elements);
```

“array”参数提供数据，而“elements”参数为数组中元素的数量。注意：数组创建后，数组大小无法更改。

有一个函数来创建与设备相关的 `seq_file`：

```
struct dentry *debugfs_create_devm_seqfile(struct device *dev,
                                             const char *name,
                                             struct dentry *parent,
                                             int (*read_fn)(struct seq_file *s,
                                                            void *data));
```

“dev”参数是与此 `debugfs` 文件相关的设备，并且“`read_fn`”是一个函数指针，这个函数在打印 `seq_file` 内容的时候被回调。

还有一些其他的面向目录的函数：

```
struct dentry *debugfs_rename(struct dentry *old_dir,
                               struct dentry *old_dentry,
                               struct dentry *new_dir,
                               const char *new_name);

struct dentry *debugfs_create_symlink(const char *name,
                                      struct dentry *parent,
                                      const char *target);
```

调用 `debugfs_rename()` 将为现有的 `debugfs` 文件重命名，可能同时切换目录。`new_name` 函数调用之前不能存在；返回值为 `old_dentry`，其中包含更新的信息。可以使用 `debugfs_create_symlink()` 创建符号链接。

所有 `debugfs` 用户必须考虑的一件事是：

`debugfs` 不会自动清除在其中创建的任何目录。如果一个模块在不显式删除 `debugfs` 目录的情况下卸载模块，结果将会遗留很多野指针，从而导致系统不稳定。因此，所有 `debugfs` 用户-至少是那些可以作为模块构建的用户-必须做模块卸载的时候准备删除在此创建的所有文件和目录。一份文件可以通过以下方式删除：

```
void debugfs_remove(struct dentry *dentry);
```

`dentry` 值可以为 `NULL` 或错误值，在这种情况下，不会有任何文件被删除。

很久以前，内核开发者使用 `debugfs` 时需要记录他们创建的每个 `dentry` 指针，以便最后所有文件都可以被清理掉。但是，现在 `debugfs` 用户能调用以下函数递归清除之前创建的文件：

```
void debugfs_remove_recursive(struct dentry *dentry);
```

如果将对应顶层目录的 `dentry` 传递给以上函数，则该目录下的整个层次结构将会被删除。

注释：[1] <http://lwn.net/Articles/309298/>

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

**Original** Documentation/filesystems/tmpfs.rst

translated by Wang Qing<[wangqing@vivo.com](mailto:wangqing@vivo.com)>

## Tmpfs

Tmpfs 是一个将所有文件都保存在虚拟内存中的文件系统。

tmpfs 中的所有内容都是临时的，也就是说没有任何文件会在硬盘上创建。如果卸载 tmpfs 实例，所有保存在其中的文件都会丢失。

tmpfs 将所有文件保存在内核缓存中，随着文件内容增长或缩小可以将不需要的页面 swap 出去。它具有最大限制，可以通过“mount -o remount …”调整。

和 ramfs (创建 tmpfs 的模板) 相比，tmpfs 包含交换和限制检查。和 tmpfs 相似的另一个东西是 RAM 磁盘 (`/dev/ram*`)，可以在物理 RAM 中模拟固定大小的硬盘，并在此之上创建一个普通的文件系统。Ramdisks 无法 swap，因此无法调整它们的大小。

由于 tmpfs 完全保存于页面缓存和 swap 中，因此所有 tmpfs 页面将在`/proc/meminfo` 中显示为“Shmem”，而在 `free(1)` 中显示为“Shared”。请注意，这些计数还包括共享内存 (shmem，请参阅 `ipcs(1)`)。获得计数的最可靠方法是使用 `df(1)` 和 `du(1)`。

tmpfs 具有以下用途：

1) 内核总有一个无法看到的内部挂载，用于共享匿名映射和 SYSV 共享内存。

挂载不依赖于 `CONFIG_TMPFS`。如果 `CONFIG_TMPFS` 未设置，tmpfs 对用户不可见。但是内部机制始终存在。

2) glibc 2.2 及更高版本期望将 tmpfs 挂载在`/dev/shm` 上以用于 POSIX 共享内存 (`shm_open`, `shm_unlink`)。添加内容到`/etc/fstab` 应注意如下：

```
tmpfs /dev/shm tmpfs defaults 0 0
```

使用时需要记住创建挂载 tmpfs 的目录。

SYSV 共享内存无需挂载，内部已默认支持。(在 2.3 内核版本中，必须挂载 tmpfs 的前身 (shm fs) 才能使用 SYSV 共享内存)

- 3) 很多人（包括我）都觉的在/tmp 和/var/tmp 上挂载非常方便，并具有较大的 swap 分区。目前循环挂载 tmpfs 可以正常工作，所以大多数发布都应当可以使用 mkinitrd 通过/tmp 访问/tmp。
- 4) 也许还有更多我不知道的地方:-)

tmpfs 有三个用于调整大小的挂载选项：

size	tmpfs 实例分配的字节数限制。默认值是不 swap 时物理 RAM 的一半。如果 tmpfs 实例过大，机器将死锁，因为 OOM 处理将无法释放该内存。
nr_blocks	与 size 相同，但以 PAGE_SIZE 为单位。
nr_inodes	tmpfs 实例的最大 inode 个数。默认值是物理内存页数的一半，或者（有高端内存的机器）低端内存 RAM 的页数，二者以较低者为准。

这些参数接受后缀 k, m 或 g 表示千，兆和千兆字节，可以在 remount 时更改。size 参数也接受后缀%用来限制 tmpfs 实例占用物理 RAM 的百分比：未指定 size 或 nr\_blocks 时，默认值为 size=50%

如果 nr\_blocks=0 (或 size=0)，block 个数将不受限制；如果 nr\_inodes=0，inode 个数将不受限制。这样挂载通常是不明智的，因为它允许任何具有写权限的用户通过访问 tmpfs 耗尽机器上的所有内存；但同时这样做也会增强在多个 CPU 的场景下的访问。

tmpfs 具有为所有文件设置 NUMA 内存分配策略挂载选项 (如果启用了 CONFIG\_NUMA)，可以通过“mount -o remount …”调整

mpol=default	采用进程分配策略 (请参阅 set_mempolicy(2))
mpol=prefer:Node	倾向从给定的节点分配
mpol=bind:NodeList	只允许从指定的链表分配
mpol=interleave	倾向于依次从每个节点分配
mpol=interleave:NodeList	依次从每个节点分配
mpol=local	优先本地节点分配内存

NodeList 格式是以逗号分隔的十进制数字表示大小和范围，最大和最小范围是用- 分隔符的十进制数来表示。例如，mpol=bind0-3,5,7,9-15

带有有效 NodeList 的内存策略将按指定格式保存，在创建文件时使用。当任务在该文件系统上创建文件时，会使用到挂载时的内存策略 NodeList 选项，如果设置的话，由调用任务的 cpuset[请参见 Documentation/admin-guide/cgroup-v1/cpusets.rst] 以及下面列出的可选标志约束。如果 NodeLists 为设置为空集，则文件的内存策略将恢复为“默认”策略。

NUMA 内存分配策略有可选标志，可以用于模式结合。在挂载 tmpfs 时指定这些可选标志可以在 NodeList 之前生效。Documentation/admin-guide/mm/numa\_memory\_policy.rst 列出所有可用的内存分配策略模式标志及其对内存策略。

=static	相当于	MPOL_F_STATIC_NODES
=relative	相当于	MPOL_F_RELATIVE_NODES

例如，mpol=bind=staticNodeList 相当于 MPOL\_BIND|MPOL\_F\_STATIC\_NODES 的分配策略

请注意，如果内核不支持 NUMA，那么使用 mpol 选项挂载 tmpfs 将会失败；nodelist 指定不在线的节点也会失败。如果您的系统依赖于此，但内核会运行不带 NUMA 功能（也许是安全 recovery 内核），或者具有较少的节点在线，建议从自动模式中省略 mpol 选项挂载选项。可以在以后通过“mount -o remount,mpol=Policy:NodeList MountPoint”添加到挂载点。

要指定初始根目录，可以使用如下挂载选项：

模式	权限用八进制数字表示
uid	用户 ID
gid	组 ID

这些选项对 remount 没有任何影响。您可以通过 chmod(1), chown(1) 和 chgrp(1) 的更改已经挂载的参数。

tmpfs 具有选择 32 位还是 64 位 inode 的挂载选项：

inode64	使用 64 位 inode
inode32	使用 32 位 inode

在 32 位内核上，默认是 inode32，挂载时指定 inode64 会被拒绝。在 64 位内核上，默认配置是 CONFIG\_TMPFS\_INODE64。inode64 避免了单个设备上可能有多个具有相同 inode 编号的文件；比如 32 位应用程序使用 glibc 如果长期访问 tmpfs，一旦达到 32 位 inode 编号，就有 EOVERRFLOW 失败的危险，无法打开大于 2GiB 的文件，并返回 EINVAL。

所以‘mount -t tmpfs -o size=10G,nr\_inodes=10k,mode=700 tmpfs /mytmpfs’将在 /mytmpfs 上挂载 tmpfs 实例，分配只能由 root 用户访问的 10GB RAM/SWAP，可以有 10240 个 inode 的实例。

作者 Christoph Rohland <cr@sap.com>, 1.12.01

更新 Hugh Dickins, 4 June 2007

更新 KOSAKI Motohiro, 16 Mar 2010

更新 Chris Down, 13 July 2020

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的

帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

### Original Documentation/scheduler/index.rst

翻译 司 延 腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)> 唐 艺 舟 Tang Yizhou <[tangyeechou@gmail.com](mailto:tangyeechou@gmail.com)>

校译

## \* Linux 调度器

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

### Original Documentation/scheduler/completion.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译 唐艺舟 Tang Yizhou <[tangyeechou@gmail.com](mailto:tangyeechou@gmail.com)>

## 完成 - “等待完成” 屏障应用程序接口 (API)

### 简介:

如果你有一个或多个线程必须等待某些内核活动达到某个点或某个特定的状态，完成可以为这个问题提供一个无竞争的解决方案。从语义上讲，它们有点像 `pthread_barrier()`，并且使用的案例类似

完成是一种代码同步机制，它比任何滥用锁/信号量和忙等待循环的行为都要好。当你想用 `yield()` 或一些古怪的 `msleep(1)` 循环来允许其它代码继续运行时，你可能想用 `wait_for_completion*()` 调用和 `completion()` 来代替。

使用“完成”的好处是，它们有一个良好定义、聚焦的目标，这不仅使得我们很容易理解代码的意图，而且它们也会生成更高效的代码，因为所有线程都可以继续执行，直到真正需要结果的时刻。而且等待和信号都高效的使用了低层调度器的睡眠/唤醒设施。

完成是建立在 Linux 调度器的等待队列和唤醒基础设施之上的。等待队列中的线程所等待的事件被简化为 `struct completion` 中的一个简单标志，被恰如其名地称为‘done’。

由于完成与调度有关，代码可以在 `kernel/sched/completion.c` 中找到。

## 用法:

使用完成需要三个主要部分:

- ‘struct completion’ 同步对象的初始化
- 通过调用 `wait_for_completion()` 的一个变体来实现等待部分。
- 通过调用 `complete()` 或 `complete_all()` 实现发信端。

也有一些辅助函数用于检查完成的状态。请注意，虽然必须先做初始化，但等待和信号部分可以按任何时间顺序出现。也就是说，一个线程在另一个线程检查是否需要等待它之前，已经将一个完成标记为“done”，这是完全正常的。

要使用完成 API，你需要 `#include <linux/completion.h>` 并创建一个静态或动态的 `struct completion` 类型的变量，它只有两个字段:

```
struct completion {
    unsigned int done;
    wait_queue_head_t wait;
};
```

结构体提供了`->wait` 等待队列来放置任务进行等待（如果有的话），以及`->done` 完成标志来表明它是否完成。完成的命名应当与正在被同步的事件名一致。一个好的例子是:

```
wait_for_completion(&early_console_added);
complete(&early_console_added);
```

好的、直观的命名（一如既往地）有助于代码的可读性。将一个完成命名为 `complete` 是没有帮助的，除非其目的是超级明显的…

## 初始化完成:

动态分配的完成对象最好被嵌入到数据结构中，以确保在函数/驱动的生命周期内存活，以防止与异步 `complete()` 调用发生竞争。

在使用 `wait_for_completion()` 的 `_timeout()` 或 `_killable()/_interruptible()` 变体时应特别小心，因为必须保证在所有相关活动（`complete()` 或 `reinit_completion()`）发生之前不会发生内存解除分配，即使这些等待函数由于超时或信号触发而过早返回。

动态分配的完成对象的初始化是通过调用 `init_completion()` 来完成的:

```
init_completion(&dynamic_object->done);
```

在这个调用中，我们初始化 `waitqueue` 并将 `->done` 设置为 0，即“not completed”或“not done”。重新初始化函数 `reinit_completion()`，只是将`->done` 字段重置为 0（“not done”），而不触及等待队列。这个函数的调用者必须确保没有任何令人讨厌的 `wait_for_completion()` 调用在并行进行。

在同一个完成对象上调用 `init_completion()` 两次很可能是一个 bug，因为它将队列重新初始化为一个空队列，已排队的任务可能会“丢失” - 在这种情况下使用 `reinit_completion()`，但要注意其他竞争。

对于静态声明和初始化，可以使用宏。

对于文件范围内的静态（或全局）声明，你可以使用 `DECLARE_COMPLETION()`:

```
static DECLARE_COMPLETION(setup_done);  
DECLARE_COMPLETION(setup_done);
```

注意，在这种情况下，完成在启动时（或模块加载时）被初始化为“not done”，不需要调用 `init_completion()`。

当完成被声明为一个函数中的局部变量时，那么应该总是明确地使用 `DECLARE_COMPLETION_ONSTACK()` 来初始化，这不仅仅是为了让 lockdep 正确运行，也是明确表名它有限的使用范围是有意为之并被仔细考虑的：

```
DECLARE_COMPLETION_ONSTACK(setup_done)
```

请注意，当使用完成对象作为局部变量时，你必须敏锐地意识到函数堆栈的短暂生命期：在所有活动（如等待的线程）停止并且完成对象完全未被使用之前，函数不得返回到调用上下文。

再次强调这一点：特别是在使用一些具有更复杂结果的等待 API 变体时，比如超时或信号（`_timeout()`, `_killable()` 和 `_interruptible()`）变体，等待可能会提前完成，而对象可能仍在被其他线程使用 - 从 `wait_on_completion*()` 调用者函数的返回会取消分配函数栈，如果 `complete()` 在其它某线程中完成调用，会引起微小的数据损坏。简单的测试可能不会触发这些类型的竞争。

如果不确定的话，使用动态分配的完成对象，最好是嵌入到其它一些生命周期长的对象中，长到超过使用完成对象的任何辅助线程的生命周期，或者有一个锁或其他同步机制来确保 `complete()` 不会在一个被释放的对象中调用。

在堆栈上单纯地调用 `DECLARE_COMPLETION()` 会触发一个 lockdep 警告。

### 等待完成:

对于一个线程来说，要等待一些并发活动的完成，它要在初始化的完成结构体上调用 `wait_for_completion()`:

```
void wait_for_completion(struct completion *done)
```

一个典型的使用场景是：

CPU#1	CPU#2
struct completion setup_done;	
init_completion(&setup_done);	
initialize_work(...,&setup_done,...);	
/* run non-dependent code */	/* do setup */

```
wait_for_completion(&setup_done);      complete(setup_done);
```

这并不意味着调用 `wait_for_completion()` 和 `complete()` 有任何特定的时间顺序-如果调用 `complete()` 发生在调用 `wait_for_completion()` 之前，那么等待方将立即继续执行，因为所有的依赖都得到了满足；如果没有，它将阻塞，直到 `complete()` 发出完成的信号。

注意，`wait_for_completion()` 是在调用 `spin_lock_irq()`/`spin_unlock_irq()`，所以只有当你知道中断被启用时才能安全地调用它。从 IRQs-off 的原子上下文中调用它将导致难以检测的错误的中断启用。

默认行为是不带超时的等待，并将任务标记为“UNINTERRUPTIBLE”状态。`wait_for_completion()` 及其变体只有在进程上下文中才是安全的（因为它们可以休眠），但在原子上下文、中断上下文、IRQ 被禁用或抢占被禁用的情况下是不安全的-关于在原子/中断上下文中处理完成的问题，还请看下面的 `try_wait_for_completion()`。

由于 `wait_for_completion()` 的所有变体都可能（很明显）阻塞很长时间，这取决于它们所等待的活动的性质，所以在大多数情况下，你可能不想在持有 `mutex` 锁的情况下调用它。

### **wait\_for\_completion\*() 可用的变体：**

下面的变体都会返回状态，在大多数（/所有）情况下都应该检查这个状态-在故意不检查状态的情况下，你可能要做一个说明（例如，见 `arch/arm/kernel/smp.c:__cpu_up()`）。

一个常见的问题是不准确的返回类型赋值，所以要注意将返回值赋值给适当类型的变量。

检查返回值的具体含义也可能被发现是相当不准确的，例如，像这样的构造：

```
if (!wait_for_completion_interruptible_timeout(...))
```

…会在成功完成和中断的情况下执行相同的代码路径-这可能不是你想要的结果：

```
int wait_for_completion_interruptible(struct completion *done)
```

这个函数在任务等待时标记为 `TASK_INTERRUPTIBLE`。如果在等待期间收到信号，它将返回 `-ERESTARTSYS`；否则为 0：

```
unsigned long wait_for_completion_timeout(struct completion *done, unsigned long timeout)
```

该任务被标记为 `TASK_UNINTERRUPTIBLE`，并将最多超时等待“timeout”个 jiffies。如果超时发生，则返回 0，否则返回剩余的时间（但至少是 1）。

超时最好用 `msecs_to_jiffies()` 或 `usecs_to_jiffies()` 计算，以使代码在很大程度上不受 HZ 的影响。

如果返回的超时值被故意忽略，那么注释应该解释原因（例如，见 `drivers/mfd/wm8350-core.c` `wm8350_read_auxadc()`）：

```
long wait_for_completion_interruptible_timeout(struct completion *done, unsigned long timeout)
```

这个函数传递一个以 jiffies 为单位的超时，并将任务标记为 TASK\_INTERRUPTIBLE。如果收到信号，则返回-ERESTARTSYS；否则，如果完成超时，则返回 0；如果完成了，则返回剩余的时间（jiffies）。

更多的变体包括 \_killable，它使用 TASK\_KILLABLE 作为指定的任务状态，如果它被中断，将返回-ERESTARTSYS，如果完成了，则返回 0。它也有一个 \_timeout 变体：

```
long wait_for_completion_killable(struct completion *done)
long wait_for_completion_killable_timeout(struct completion *done, unsigned long timeout)
```

`wait_for_completion_io()` 的 \_io 变体的行为与非 \_io 变体相同，只是将等待时间计为“IO 等待”，这对任务在调度/IO 统计中的计算方式有影响：

```
void wait_for_completion_io(struct completion *done)
unsigned long wait_for_completion_io_timeout(struct completion *done, unsigned long timeout)
```

## 对完成发信号：

一个线程想要发出信号通知继续的条件已经达到，就会调用 `complete()`，向其中一个等待者发出信号表明它可以继续：

```
void complete(struct completion *done)
```

…or calls `complete_all()` to signal all current and future waiters:

```
void complete_all(struct completion *done)
```

即使在线程开始等待之前就发出了完成的信号，信号传递也会继续进行。这是通过等待者“consuming”（递减）“struct completion”的完成字段来实现的。等待的线程唤醒的顺序与它们被排队的顺序相同（FIFO 顺序）。

如果多次调用 `complete()`，那么这将允许该数量的等待者继续进行—每次调用 `complete()` 将简单地增加已完成的字段。但多次调用 `complete_all()` 是一个错误。`complete()` 和 `complete_all()` 都可以在 IRQ/atomic 上下文中安全调用。

在任何时候，只能有一个线程在一个特定的“`struct completion`”上调用 `complete()` 或 `complete_all()` - 通过等待队列自旋锁进行序列化。任何对 `complete()` 或 `complete_all()` 的并发调用都可能是一个设计错误。

从 IRQ 上下文中发出完成信号是可行的，因为它将正确地用 `spin_lock_irqsave()/spin_unlock_irqrestore()` 执行锁操作

## try\_wait\_for\_completion()/completion\_done():

`try_wait_for_completion()` 函数不会将线程放在等待队列中，而是在需要排队（阻塞）线程时返回 `false`，否则会消耗一个已发布的完成并返回 `true`:

```
bool try_wait_for_completion(struct completion *done)
```

最后，为了在不以任何方式改变完成的情况下检查完成的状态，可以调用 `completion_done()`，如果没有发布的完成尚未被等待者消耗，则返回 `false`（意味着存在等待者），否则返回 `true`:

```
bool completion_done(struct completion *done)
```

`try_wait_for_completion()` 和 `completion_done()` 都可以在 IRQ 或原子上下文中安全调用。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

**Original** Documentation/scheduler/sched-arch.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译

## 架构特定代码的 CPU 调度器实现提示

Nick Piggin, 2005

## 上下文切换

1. 运行队列锁默认情况下，`switch_to_arch` 函数在调用时锁定了运行队列。这通常不是一个间题，除非 `switch_to` 可能需要获取运行队列锁。这通常是由于上下文切换中的唤醒操作造成的。见 `arch/ia64/include/asm/switch_to.h` 的例子。

为了要求调度器在运行队列解锁的情况下调用 `switch_to`，你必须在头文件中`#define \_\_ARCH\_WANT\_UNLOCKED\_CTXSW`（通常是定义 `switch_to` 的那个文件）。

在 `CONFIG_SMP` 的情况下，解锁的上下文切换对核心调度器的实现只带来了非常小的性能损失。

### CPU 空转

你的 cpu\_idle 程序需要遵守以下规则：

1. 现在抢占应该在空闲的例程上禁用。应该只在调用 schedule() 时启用，然后再禁用。
2. need\_resched/TIF\_NEED\_RESCHED 只会被设置，并且在运行任务调用 schedule() 之前永远不会被清除。空闲线程只需要查询 need\_resched，并且永远不会设置或清除它。
3. 当 cpu\_idle 发现 (need\_resched() == ‘true’ )，它应该调用 schedule()。否则它不应该调用 schedule()。
4. 在检查 need\_resched 时，唯一需要禁用中断的情况是，我们要让处理器休眠到下一个中断（这并不对 need\_resched 提供任何保护，它可以防止丢失一个中断）：
  - 4a. 这种睡眠类型的常见问题似乎是：

```
local_irq_disable();
if (!need_resched()) {
    local_irq_enable();
    *** resched interrupt arrives here ***
    __asm__("sleep until next interrupt");
}
```

5. 当 need\_resched 变为高电平时，TIF\_POLLING\_NRFLAG 可以由不需要中断来唤醒它们的空闲程序设置。换句话说，它们必须定期轮询 need\_resched，尽管做一些后台工作或进入低 CPU 优先级可能是合理的。
  - 5a. 如果 TIF\_POLLING\_NRFLAG 被设置，而我们确实决定进入一个中断睡眠，那么需要清除它，然后发出一个内存屏障（接着测试 need\_resched，禁用中断，如 3 中解释）。

arch/x86/kernel/process.c 有轮询和睡眠空闲函数的例子。

### 可能出现的 arch/问题

我发现的可能的 arch 问题（并试图解决或没有解决）。：

ia64 - safe\_halt 的调用与中断相比，是否很荒谬？（它睡眠了吗）（参考 #4a）

sh64 - 睡眠与中断相比，是否很荒谬？（参考 #4a）

sparc - 在这一点上，IRQ 是开着的（?），把 local\_irq\_save 改为 \_disable。

- 待办事项：需要第二个 CPU 来禁用抢占（参考 #1）

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题, 请联系该文件的译者, 或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

### Original Documentation/scheduler/sched-bwc.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译

## CFS 带宽控制

---

**Note:** 本文只讨论了 SCHED\_NORMAL 的 CPU 带宽控制。SCHED\_RT 的情况在 Documentation/scheduler/sched-rt-group.rst 中有涉及。

---

CFS 带宽控制是一个 CONFIG\_FAIR\_GROUP\_SCHED 扩展, 它允许指定一个组或层次的最大 CPU 带宽。一个组允许的带宽是用配额和周期指定的。在每个给定的”周期“（微秒）内, 一个任务组被分配多达“配额”微秒的 CPU 时间。当 cgroup 中的线程可运行时, 该配额以时间片段的方式被分配到每个 cpu 运行队列中。一旦所有的配额被分配, 任何额外的配额请求将导致这些线程被限流。被限流的线程将不能再次运行, 直到下一个时期的配额得到补充。

一个组的未分配配额是全局跟踪的, 在每个周期边界被刷新为 cfs\_quota 单元。当线程消耗这个带宽时, 它以需求为基础被转移到 cpu-local “简仓”, 在每次更新中转移的数量是可调整的, 被描述为“片”(时间片)。

## 突发特性

现在这个功能借来的时间是用于防范我们对未来的低估, 代价是对其他系统用户的干扰增加。所有这些都有很好的限制。

传统的 (UP-EDF) 带宽控制是这样的:

$$(U = \text{Sum } u_i) \leq 1$$

这既保证了每个最后期限的实现, 也保证了系统的稳定。毕竟, 如果  $U > 1$ , 那么每一秒钟的壁钟时间, 我们就必须运行超过一秒钟的程序时间, 显然会错过我们的最后期限, 但下一个最后期限会更远, 永远没有时间赶上, 无边无界的失败。

突发特性观察到工作负载并不总是执行全部配额; 这使得人们可以将  $u_i$  描述为一个统计分布。

例如, 让  $u_i = \{x, e\}_i$ , 其中  $x$  是  $p(95)$  和  $x+e p(100)$  (传统的 WCET)。这实际上允许  $u$  更小, 提高了效率 (我们可以在系统中打包更多的任务), 但代价是当所有的概率都一致时, 会错过最后期限。然而, 它确实保持了稳定性, 因为只要我们的  $x$  高于平均水平, 每一次超限都必须与低估相匹配。

也就是说, 假设我们有两个任务, 都指定了一个  $p(95)$  值, 那么我们有一个  $p(95)*p(95)=90.25\%$  的机会, 两个任务都在他们的配额内, 一切都很好。同时, 我们有一个  $p(5)p(5)=0.25\%$  的机会, 两个任务同时超过

他们的配额（保证最后期限失败）。在这两者之间有一个阈值，其中一个超过了，而另一个没有不足，无法补偿；这取决于具体的 CDFs。

同时，我们可以说，最坏的情况下截断日期失败，将是  $\text{Sum } e_i$ ；也就是说，有一个有界的迟延（在假设  $x+e$  确实是 WCET 的情况下）。

使用突发时的干扰是由错过最后期限的可能性和平均 WCET 来评价的。测试结果表明，当有许多 cgroup 或 CPU 未被充分利用时，干扰是有限的。更多的细节显示在：<https://lore.kernel.org/lkml/5371BD36-55AE-4F71-B9D7-B86DC32E3D2B@linux.alibaba.com/>

## 管理

配额、周期和突发是在 cpu 子系统内通过 cgroupfs 管理的。

---

**Note:** 本节描述的 cgroupfs 文件只适用于 cgroup v1。对于 cgroup v2，请参阅 Control Group v2。Documentation/admin-guide/cgroup-v2.rst.

---

- `cpu.cfs_quota_us`: 在一个时期内补充的运行时间（微秒）。
- `cpu.cfs_period_us`: 一个周期的长度（微秒）。
- `cpu.stat`: 输出节流统计数据 [下面进一步解释]
- `cpu.cfs_burst_us`: 最大累积运行时间（微秒）。

默认值是：

```
cpu.cfs_period_us=100ms
cpu.cfs_quota_us=-1
cpu.cfs_burst_us=0
```

`cpu.cfs_quota_us` 的值为 -1 表示该组没有任何带宽限制，这样的组被描述为无限制的带宽组。这代表了 CFS 的传统工作保护行为。

写入不小于 `cpu.cfs_burst_us` 的任何（有效的）正值将配发指定的带宽限制。该配额或周期允许的最小配额是 1ms。周期长度也有一个 1s 的上限。当带宽限制以分层方式使用时，存在额外的限制，这些在下面有更详细的解释。

向 `cpu.cfs_quota_us` 写入任何负值都会移除带宽限制，并使组再次回到无限制的状态。

`cpu.cfs_burst_us` 的值为 0 表示该组不能积累任何未使用的带宽。它使得 CFS 的传统带宽控制行为没有改变。将不大于 `cpu.cfs_quota_us` 的任何（有效的）正值写入 `cpu.cfs_burst_us` 将配发未使用带宽累积的上限。

如果一个组处于受限状态，对该组带宽规格的任何更新都将导致其成为无限流状态。

## 系统范围设置

为了提高效率，运行时间在全局池和 CPU 本地“筒仓”之间以批处理方式转移。这大大减少了大型系统的全局核算压力。每次需要进行这种更新时，传输的数量被描述为“片”。

这是可以通过 procfs 调整的：

```
/proc/sys/kernel/sched_cfs_bandwidth_slice_us (default=5ms)
```

较大的时间片段值将减少传输开销，而较小的值则允许更精细的消费。

## 统计

一个组的带宽统计数据通过 `cpu.stat` 的 5 个字段导出。

`cpu.stat`:

- `nr_periods`: 已经过去的执行间隔的数量。
- `nr_throttled`: 该组已被节流/限制的次数。
- `throttled_time`: 该组的实体被限流的总时间长度（纳秒）。
- `nr_bursts`: 突发发生的周期数。
- `burst_time`: 任何 CPU 在各个时期使用超过配额的累计壁钟时间（纳秒）。

这个接口是只读的。

## 分层考虑

该接口强制要求单个实体的带宽总是可以达到的，即： $\max(c_i) \leq C$ 。然而，在总体情况下，是明确允许过度订阅的，以便在一个层次结构中实现工作保护语义：

例如， $\text{Sum } (c_i)$  可能超过 C

[ 其中 C 是父方的带宽， $c_i$  是其子方的带宽。 ]

**Note:** 译文中的父亲/孩子指的是 cgroup parent, cgroup children。

有两种方式可以使一个组变得限流：

- a. 它在一段时期内完全消耗自己的配额
- b. 父方的配额在其期间内全部用完

在上述 b) 情况下，即使孩子可能有剩余的运行时间，它也不会被允许，直到父亲的运行时间被刷新。

## CFS 带宽配额的注意事项

一旦一个片断被分配给一个 cpu，它就不会过期。然而，如果该 cpu 上的所有线程都无法运行，那么除了 1ms 以外的所有时间片都可以返回到全局池中。这是在编译时由 `min_cfs_rq_runtime` 变量配置的。这是一个性能调整，有助于防止对全局锁的额外争夺。

`cpu-local` 分片不会过期的事实导致了一些有趣的罕见案例，应该被理解。

对于 cgroup cpu 限制的应用程序来说，这是一个相对有意义的问题，因为他们自然会消耗他们的全部配额，以及每个 cpu-本地片在每个时期的全部。因此，预计 `nr_periods` 大致等于 `nr_throttled`，并且 `cpuacct` 用量的增加大致等于 `cfs_quota_us` 在每个周期的增加。

对于高线程、非 cpu 绑定的应用程序，这种非过期的细微差别允许应用程序短暂地突破他们的配额限制，即任务组正在运行的每个 cpu 上未使用的片断量（通常每个 cpu 最多 1ms 或由 `min_cfs_rq_runtime` 定义）。这种轻微的突发只适用于配额已经分配给 cpu，然后没有完全使用或在以前的时期返回。这个突发量不会在核心之间转移。因此，这种机制仍然严格限制任务组的配额平均使用量，尽管是在比单一时期更长的时间窗口。这也限制了突发能力，每个 cpu 不超过 1ms。这为在高核数机器上有小配额限制的高线程应用提供了更好的更可预测的用户体验。它还消除了在使用低于配额的 cpu 时对这些应用进行节流的倾向。另一种说法是，通过允许一个片断的未使用部分在不同时期保持有效，我们减少了在不需要整个片断的 cpu 时间的 `cpu-local` 简仓上浪费配额的可能性。

绑定 cpu 和非绑定 cpu 的交互式应用之间的互动也应该被考虑，特别是当单核使用率达到 100% 时。如果你给了这些应用程序一半的 cpu-core，并且它们都被安排在同一个 CPU 上，理论上非 cpu 绑定的应用程序有可能在某些时期使用多达 1ms 的额外配额，从而阻止 cpu 绑定的应用程序完全使用其配额，这也是同样的数量。在这些情况下，将由 CFS 算法（见 CFS 调度器）来决定选择哪个应用程序来运行，因为它们都是可运行的，并且有剩余的配额。这个运行时间的差异将在接下来的交互式应用程序空闲期间得到弥补。

## 例子

1. 限制一个组的运行时间为 1 个 CPU 的价值:

如果周期是 250ms，配额也是 250ms，那么该组将每 250ms 获得价值 1 个 CPU 的运行时间。

```
# echo 250000 > cpu.cfs_quota_us /* quota = 250ms */
# echo 250000 > cpu.cfs_period_us /* period = 250ms */
```

2. 在多 CPU 机器上，将一个组的运行时间限制为 2 个 CPU 的价值

在 500ms 周期和 1000ms 配额的情况下，该组每 500ms 可以获得 2 个 CPU 的运行时间：

```
# echo 1000000 > cpu.cfs_quota_us /* quota = 1000ms */
# echo 500000 > cpu.cfs_period_us /* period = 500ms */
```

这里较大的周期允许增加突发能力。

3. 将一个组限制在 1 个 CPU 的 20%。

在 50ms 周期内，10ms 配额将相当于 1 个 CPU 的 20%。：

```
# echo 10000 > cpu.cfs_quota_us /* quota = 10ms */
# echo 50000 > cpu.cfs_period_us /* period = 50ms */
```

通过在这里使用一个小的周期，我们以牺牲突发容量为代价来确保稳定的延迟响应。

- 将一个组限制在 1 个 CPU 的 40%，并允许累积到 1 个 CPU 的 20%，如果已经累积了的话。

在 50ms 周期内，20ms 配额将相当于 1 个 CPU 的 40%。而 10 毫秒的突发将相当于 1 个 CPU 的 20%：

```
# echo 20000 > cpu.cfs_quota_us /* quota = 20ms */
# echo 50000 > cpu.cfs_period_us /* period = 50ms */
# echo 10000 > cpu.cfs_burst_us /* burst = 10ms */
```

较大的缓冲区设置（不大于配额）允许更大的突发容量。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:<alexs@kernel.org>)。

**Original** Documentation/scheduler/sched-design-CFS.rst

翻译 唐艺舟 Tang Yizhou <[tangyeechou@gmail.com](mailto:tangyeechou@gmail.com)>

## 完全公平调度器

### 1. 概述

CFS 表示“完全公平调度器”，它是为桌面新设计的进程调度器，由 Ingo Molnar 实现并合入 Linux 2.6.23。它替代了之前原始调度器中 SCHED\_OTHER 策略的交互式代码。

CFS 80% 的设计可以总结为一句话：CFS 在真实硬件上建模了一个“理想的，精确的多任务 CPU”。

“理想的多任务 CPU”是一种（不存在的:-)）具有 100% 物理算力的 CPU，它能让每个任务精确地以相同的速度并行运行，速度均为  $1/nr\_running$ 。举例来说，如果有两个任务正在运行，那么每个任务获得 50% 物理算力。—也就是说，真正的并行。

在真实的硬件上，一次只能运行一个任务，所以我们需要介绍“虚拟运行时间”的概念。任务的虚拟运行时间表明，它的下一个时间片将在上文描述的理想多任务 CPU 上开始执行。在实践中，任务的虚拟运行时间由它的真实运行时间相较于正在运行的任务总数归一化计算得到。

### 2. 一些实现细节

在 CFS 中，虚拟运行时间由每个任务的 `p->se.vruntime`（单位为纳秒）的值表达和跟踪。因此，精确地计时和测量一个任务应得的“预期的 CPU 时间”是可能的。

一些细节：在“理想的”硬件上，所有的任务在任何时刻都应该具有一样的 `p->se.vruntime` 值，—也就是说，任务应当同时执行，没有任务会在“理想的”CPU 分时中变得“不平衡”。

CFS 的任务选择逻辑基于 `p->se.vruntime` 的值，因此非常简单：总是试图选择 `p->se.vruntime` 值最小的任务运行（也就是说，至今执行时间最少的任务）。CFS 总是尽可能尝试按“理想多任务硬件”那样将 CPU 时间在可运行任务中均分。

CFS 剩下的其它设计，一般脱离了这个简单的概念，附加的设计包括 nice 级别，多处理，以及各种用来识别已睡眠任务的算法变体。

### 3. 红黑树

CFS 的设计非常激进：它不使用运行队列的旧数据结构，而是使用按时间排序的红黑树，构建出任务未来执行的“时间线”。因此没有任何“数组切换”的旧包袱（之前的原始调度器和 RSDL/SD 都被它影响）。

CFS 同样维护了 `rq->cfs.min_vruntime` 值，它是单调递增的，跟踪运行队列中的所有任务的最小虚拟运行时间值。系统做的全部工作是：使用 `min_vruntime` 跟踪，然后用它的值将新激活的调度实体尽可能地放在红黑树的左侧。

运行队列中正在运行的任务的总数由 `rq->cfs.load` 计数，它是运行队列中的任务的权值之和。

CFS 维护了一个按时间排序的红黑树，所有可运行任务以 `p->se.vruntime` 为键值排序。CFS 从这棵树上选择“最左侧”的任务并运行。系统继续运行，被执行过的任务越来越被放到树的右侧—缓慢，但很明确每个任务都有成为“最左侧任务”的机会，因此任务将确定性地获得一定量 CPU 时间。

总结一下，CFS 工作方式像这样：它运行一个任务一会儿，当任务发生调度（或者调度器时钟滴答 tick 产生），就会考虑任务的 CPU 使用率：任务刚刚花在物理 CPU 上的（少量）时间被加到 `p->se.vruntime`。一旦 `p->se.vruntime` 变得足够大，其它的任务将成为按时间排序的红黑树的“最左侧任务”（相较最左侧的任务，还要加上一个很小的“粒度”量，使得我们不会对任务过度调度，导致缓存颠簸），然后新的最左侧任务将被选中，当前任务被抢占。

### 4. CFS 的一些特征

CFS 使用纳秒粒度的计时，不依赖于任何 jiffies 或 HZ 的细节。因此 CFS 并不像之前的调度器那样有“时间片”的概念，也没有任何启发式的设计。唯一可调的参数（你需要打开 CONFIG\_SCHED\_DEBUG）是：

`/proc/sys/kernel/sched_min_granularity_ns`

它可以用来将调度器从“桌面”模式（也就是低时延）调节为“服务器”（也就是高批处理）模式。它的默认设置是适合桌面的工作负载。SCHED\_BATCH 也被 CFS 调度器模块处理。

CFS 的设计不易受到当前存在的任何针对 stock 调度器的“攻击”的影响，包括 fifty.c, thud.c, chew.c, ring-test.c, massive\_intr.c，它们都能很好地运行，不会影响交互性，将产生符合预期的行为。

CFS 调度器处理 nice 级别和 SCHED\_BATCH 的能力比之前的原始调度器更强：两种类型的工作负载都被更激进地隔离了。

SMP 负载均衡被重做/清理过：遍历运行队列的假设已经从负载均衡的代码中移除，使用调度模块的迭代器。结果是，负载均衡代码变得简单不少。

## 5. 调度策略

CFS 实现了三种调度策略：

- SCHED\_NORMAL: (传统被称为 SCHED\_OTHER): 该调度策略用于普通任务。
- SCHED\_BATCH: 抢占不像普通任务那样频繁，因此允许任务运行更长时间，更好地利用缓存，不过要以交互性为代价。它很适合批处理工作。
- SCHED\_IDLE: 它比 nice 19 更弱，不过它不是真正的 idle 定时器调度器，因为要避免给机器带来死锁的优先级反转问题。

SCHED\_FIFO/\_RR 被实现在 sched/rt.c 中，它们由 POSIX 具体说明。

util-linux-ng 2.13.1.1 中的 chrt 命令可以设置以上所有策略，除了 SCHED\_IDLE。

## 6. 调度类

新的 CFS 调度器被设计成支持“调度类”，一种调度模块的可扩展层次结构。这些模块封装了调度策略细节，由调度器核心代码处理，且无需对它们做太多假设。

sched/fair.c 实现了上文描述的 CFS 调度器。

sched/rt.c 实现了 SCHED\_FIFO 和 SCHED\_RR 语义，且比之前的原始调度器更简洁。它使用了 100 个运行队列（总共 100 个实时优先级，替代了之前调度器的 140 个），且不需要过期数组（expired array）。

调度类由 sched\_class 结构体实现，它包括一些函数钩子，当感兴趣的事件发生时，钩子被调用。

这是（部分）钩子的列表：

- enqueue\_task(…)

当任务进入可运行状态时，被调用。它将调度实体（任务）放到红黑树中，增加 nr\_running 变量的值。

- dequeue\_task(…)

当任务不再可运行时，这个函数被调用，对应的调度实体被移出红黑树。它减少 nr\_running 变量的值。

- yield\_task(…)

这个函数的行为基本上是出队，紧接着入队，除非 compat\_yield sysctl 被开启。在那种情况下，它将调度实体放在红黑树的最右端。

- `check_preempt_curr(…)`

这个函数检查进入可运行状态的任务能否抢占当前正在运行的任务。

- `pick_next_task(…)`

这个函数选择接下来最适合运行的任务。

- `set_curr_task(…)`

这个函数在任务改变调度类或改变任务组时被调用。

- `task_tick(…)`

这个函数最常被时间滴答函数调用，它可能导致进程切换。这驱动了运行时抢占。

## 7. CFS 的组调度扩展

通常，调度器操作粒度为任务，努力为每个任务提供公平的 CPU 时间。有时可能希望将任务编组，并为每个组提供公平的 CPU 时间。举例来说，可能首先希望为系统中的每个用户提供公平的 CPU 时间，接下来才是某个用户的每个任务。

`CONFIG_CGROUP_SCHED` 力求实现它。它将任务编组，并为这些组公平地分配 CPU 时间。

`CONFIG_RT_GROUP_SCHED` 允许将实时（也就是说，`SCHED_FIFO` 和 `SCHED_RR`）任务编组。

`CONFIG_FAIR_GROUP_SCHED` 允许将 CFS（也就是说，`SCHED_NORMAL` 和 `SCHED_BATCH`）任务编组。

这些编译选项要求 `CONFIG_CGROUPS` 被定义，然后管理员能使用 `cgroup` 伪文件系统任意创建任务组。关于该文件系统的更多信息，参见 `Documentation/admin-guide/cgroup-v1/cgroups.rst`

当 `CONFIG_FAIR_GROUP_SCHED` 被定义后，通过伪文件系统，每个组被创建一个“`cpu.shares`”文件。参见下面的例子来创建任务组，并通过“`cgroup`”伪文件系统修改它们的 CPU 份额：

```
# mount -t tmpfs cgroup_root /sys/fs/cgroup
# mkdir /sys/fs/cgroup/cpu
# mount -t cgroup -ocpu none /sys/fs/cgroup/cpu
# cd /sys/fs/cgroup/cpu

# mkdir multimedia      # 创建 "multimedia" 任务组
# mkdir browser         # 创建 "browser" 任务组

# # 配置 multimedia 组，令其获得 browser 组两倍 CPU 带宽

# echo 2048 > multimedia/cpu.shares
# echo 1024 > browser/cpu.shares

# firefox &      # 启动 firefox 并把它移到 "browser" 组
# echo <firefox_pid> > browser/tasks
```

```
# # 启动 gmpLayer (或者你最喜欢的电影播放器)
# echo <movie_player_pid> > multimedia/tasks
```

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

### Original Documentation/scheduler/sched-domains.rst

翻译 唐艺舟 Tang Yizhou <[tangyeechou@gmail.com](mailto:tangyeechou@gmail.com)>

校译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## 调度域

每个 CPU 有一个“基”调度域（`struct sched_domain`）。调度域层次结构从基调度域构建而来，可通过`->parent` 指针自下而上遍历。`->parent` 必须以 NULL 结尾，调度域结构体必须是 per-CPU 的，因为它们无锁更新。

每个调度域管辖数个 CPU（存储在`->span` 字段中）。一个调度域的 `span` 必须是它的子调度域 `span` 的超集（如有需求出现，这个限制可以放宽）。CPU i 的基调度域必须至少管辖 CPU i。每个 CPU 的顶层调度域通常将会管辖系统中的全部 CPU，尽管严格来说这不是必须的，假如是这样，会导致某些 CPU 出现永远不会被指定任务运行的情况，直到允许的 CPU 掩码被显式设定。调度域的 `span` 字段意味着“在这些 CPU 中做进程负载均衡”。

每个调度域必须具有一个或多个 CPU 调度组（`struct sched_group`），它们以单向循环链表的形式组织，存储在`->groups` 指针中。这些组的 CPU 掩码的并集必须和调度域 `span` 字段一致。`->groups` 指针指向的这些组包含的 CPU，必须被调度域管辖。组包含的是只读数据，被创建之后，可能被多个 CPU 共享。任意两个组的 CPU 掩码的交集不一定为空，如果是这种情况，对应调度域的 `SD_OVERLAP` 标志位被设置，它管辖的调度组可能不能在多个 CPU 中共享。

调度域中的负载均衡发生在调度组中。也就是说，每个组被视为一个实体。组的负载被定义为它管辖的每个 CPU 的负载之和。仅当组的负载不均衡后，任务才在组之间发生迁移。

在 `kernel/sched/core.c` 中，`trigger_load_balance()` 在每个 CPU 上通过 `scheduler_tick()` 周期执行。在当前运行队列下一个定期调度再平衡事件到达后，它引发一个软中断。负载均衡真正的工作由 `run_rebalance_domains()->rebalance_domains()` 完成，在软中断上下文中执行 (`SCHED_SOFTIRQ`)。

后一个函数有两个入参：当前 CPU 的运行队列、它在 `scheduler_tick()` 调用时是否空闲。函数会从当前 CPU 所在的基调度域开始迭代执行，并沿着 `parent` 指针链向上进入更高层级的调度域。在迭代过程中，函

数会检查当前调度域是否已经耗尽了再平衡的时间间隔，如果是，它在该调度域运行 `load_balance()`。接下来它检查父调度域（如果存在），再后来父调度域的父调度域，以此类推。

起初，`load_balance()` 查找当前调度域中最繁忙的调度组。如果成功，在该调度组管辖的全部 CPU 的运行队列中找出最繁忙的运行队列。如能找到，对当前的 CPU 运行队列和新找到的最繁忙运行队列均加锁，并把任务从最繁忙队列中迁移到当前 CPU 上。被迁移的任务数量等于在先前迭代执行中计算出的该调度域的调度组的不均衡值。

### 实现调度域

基调度域会管辖 CPU 层次结构中的第一层。对于超线程（SMT）而言，基调度域将会管辖同一个物理 CPU 的全部虚拟 CPU，每个虚拟 CPU 对应一个调度组。

在 SMP 中，基调度域的父调度域将会管辖同一个结点中的全部物理 CPU，每个调度组对应一个物理 CPU。接下来，如果是非统一内存访问（NUMA）系统，SMP 调度域的父调度域将管辖整个机器，一个结点的 CPU 掩码对应一个调度组。亦或，你可以使用多级 NUMA；举例来说 Opteron 处理器，可能仅用一个调度域来覆盖它的一个 NUMA 层级。

实现者需要阅读 `include/linux/sched/sd_flags.h` 的注释：读 `SD_*` 来了解具体情况以及调度域的 SD 标志位调节了哪些东西。

体系结构可以把指定的拓扑层级的通用调度域构建器和默认的 SD 标志位覆盖掉，方法是创建一个 `sched_domain_topology_level` 数组，并以该数组作为入参调用 `set_sched_topology()`。

调度域调试基础设施可以通过 `CONFIG_SCHED_DEBUG` 开启，并在开机启动命令行中增加“`sched_verbose`”。如果你忘记调整开机启动命令行了，也可以打开 `/sys/kernel/debug/sched/verbose` 开关。这将开启调度域错误检查的解析，它应该能捕获（上文描述过的）绝大多数错误，同时以可视化格式打印调度域的结构。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alex@kernel.org>](mailto:<alex@kernel.org>)。

---

**Original** Documentation/scheduler/sched-capacity.rst

翻译 唐艺舟 Tang Yizhou <[tangyeechou@gmail.com](mailto:tangyeechou@gmail.com)>

校译 时奎亮 Alex Shi <[alex@kernel.org](mailto:alex@kernel.org)>

## 算力感知调度

### 1. CPU 算力

#### 1.1 简介

一般来说，同构的 SMP 平台由完全相同的 CPU 构成。异构的平台则由性能特征不同的 CPU 构成，在这样的平台中，CPU 不能被认为是相同的。

我们引入 CPU 算力 (capacity) 的概念来测量每个 CPU 能达到的性能，它的值相对系统中性能最强的 CPU 做过归一化处理。异构系统也被称为非对称 CPU 算力系统，因为它们由不同算力的 CPU 组成。

最大可达性能（换言之，最大 CPU 算力）的差异有两个主要来源：

- 不是所有 CPU 的微架构都相同。
- 在动态电压频率升降 (Dynamic Voltage and Frequency Scaling, DVFS) 框架中，不是所有的 CPU 都能达到一样高的操作性能值 (Operating Performance Points, OPP。译注，也就是“频率-电压”对)。

Arm 大小核 (big.LITTLE) 系统是同时具有两种差异的一个例子。相较小核，大核面向性能（拥有更多的流水线层级，更大的缓存，更智能的分支预测器等），通常可以达到更高的操作性能值。

CPU 性能通常由每秒百万指令 (Millions of Instructions Per Second, MIPS) 表示，也可表示为 per Hz 能执行的指令数，故：

$$\text{capacity}(\text{cpu}) = \text{work\_per\_hz}(\text{cpu}) * \text{max\_freq}(\text{cpu})$$

#### 1.2 调度器术语

调度器使用了两种不同的算力值。CPU 的 `capacity_orig` 是它的最大可达算力，即最大可达性能等级。CPU 的 `capacity` 是 `capacity_orig` 扣除了一些性能损失（比如处理中断的耗时）的值。

注意 CPU 的 `capacity` 仅仅被设计用于 CFS 调度类，而 `capacity_orig` 是不感知调度类的。为简洁起见，本文档的剩余部分将不加区分的使用术语 `capacity` 和 `capacity_orig`。

### 1.3 平台示例

#### 1.3.1 操作性能值相同

考虑一个假想的双核非对称 CPU 算力系统，其中

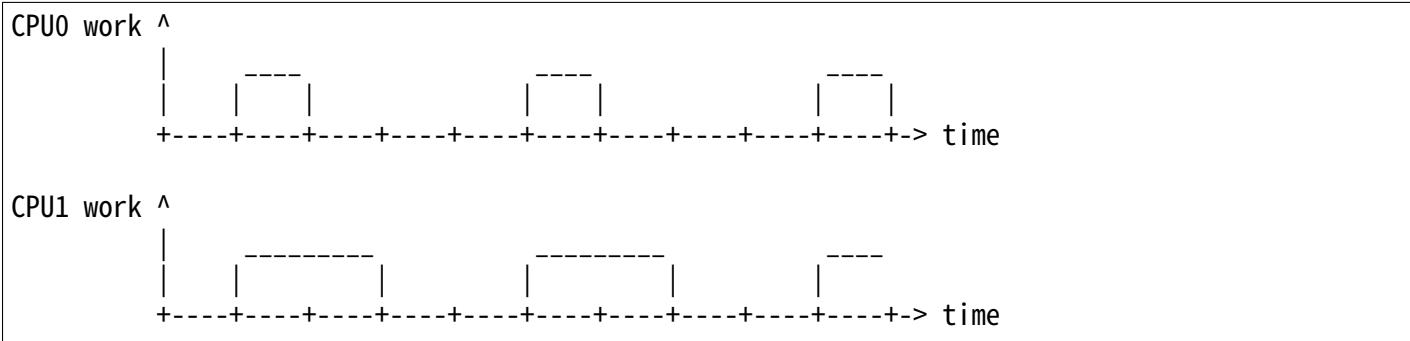
- `work_per_hz(CPU0) = W`
- `work_per_hz(CPU1) = W/2`
- 所有 CPU 以相同的固定频率运行

根据上文对算力的定义:

- $\text{capacity}(\text{CPU0}) = C$
- $\text{capacity}(\text{CPU1}) = C/2$

若这是 Arm 大小核系统, 那么 CPU0 是大核, 而 CPU1 是小核。

考虑一种周期性产生固定工作量的工作负载, 你将会得到类似下图的执行轨迹:



CPU0 在系统中具有最高算力 ( $C$ ), 它使用  $T$  个单位时间完成固定工作量  $W$ 。另一方面, CPU1 只有 CPU0 一半算力, 因此在  $T$  个单位时间内仅完成工作量  $W/2$ 。

### 1.3.2 最大操作性能值不同

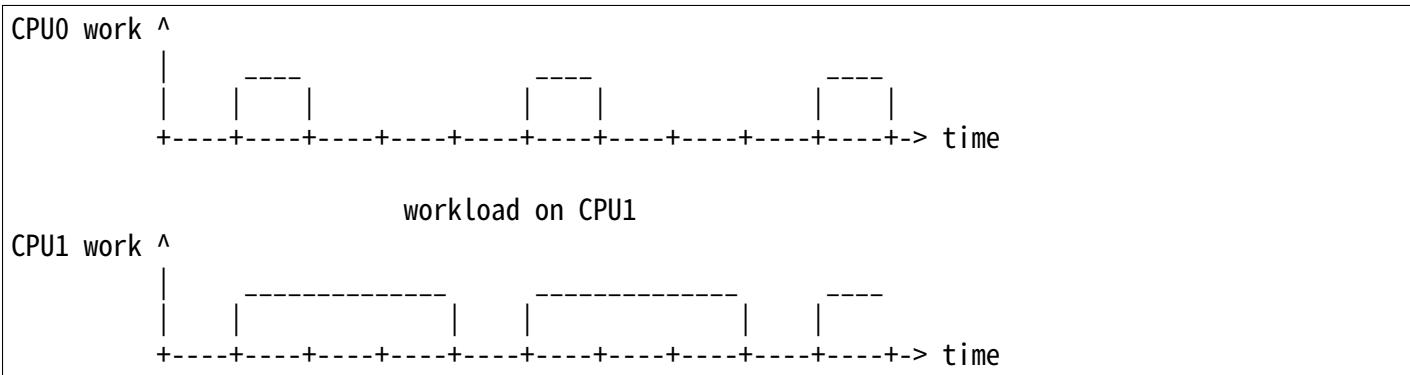
具有不同算力值的 CPU, 通常来说最大操作性能值也不同。考虑上一小节提到的 CPU (也就是说, `work_per_hz()` 相同) :

- $\text{max\_freq}(\text{CPU0}) = F$
- $\text{max\_freq}(\text{CPU1}) = 2/3 * F$

这将推出:

- $\text{capacity}(\text{CPU0}) = C$
- $\text{capacity}(\text{CPU1}) = C/3$

执行 1.3.1 节描述的工作负载, 每个 CPU 按最大频率运行, 结果为:



## 1.4 关于计算方式的注意事项

需要注意的是，使用单一值来表示 CPU 性能的差异是有些争议的。两个不同的微架构的相对性能差异应该描述为：X% 整数运算差异，Y% 浮点数运算差异，Z% 分支跳转差异，等等。尽管如此，使用简单计算方式的结果目前还是令人满意的。

## 2. 任务使用率

### 2.1 简介

算力感知调度要求描述任务需求，描述方式要和 CPU 算力相关。每个调度类可以用不同的方式描述它。任务使用率是 CFS 独有的描述方式，不过在这里介绍它有助于引入更多一般性的概念。

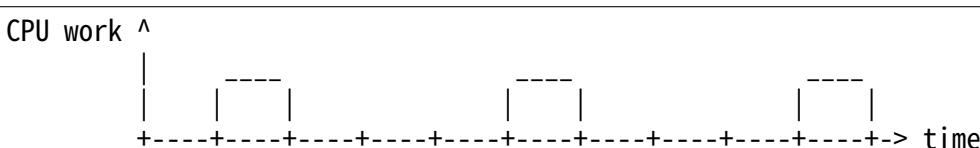
任务使用率是一种用百分比来描述任务吞吐率需求的方式。一个简单的近似是任务的占空比，也就是说：

$$\text{task\_util}(p) = \text{duty\_cycle}(p)$$

在频率固定的 SMP 系统中，100% 的利用率意味着任务是忙等待循环。反之，10% 的利用率暗示这是一个小周期任务，它在睡眠上花费的时间比执行更多。

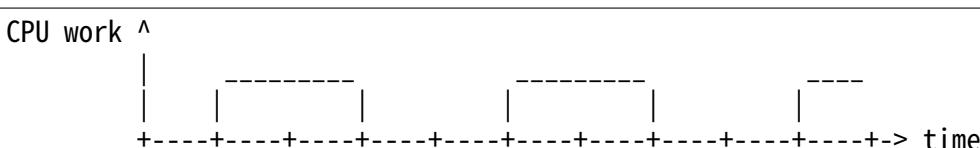
### 2.2 频率不变性

一个需要考虑的议题是，工作负载的占空比受 CPU 正在运行的操作性能值直接影响。考虑以给定的频率 F 执行周期性工作负载：



可以算出  $\text{duty\_cycle}(p) == 25\%$ 。

现在，考虑以给定频率  $F/2$  执行 同一个工作负载：



可以算出  $\text{duty\_cycle}(p) == 50\%$ ，尽管两次执行中，任务的行为完全一致（也就是说，执行的工作量相同）。

任务利用率信号可按下面公式处理成频率不变的（译注：这里的术语用到了信号与系统的概念）：

$$\text{task\_util\_freq\_inv}(p) = \text{duty\_cycle}(p) * (\text{curr\_frequency(cpu)} / \text{max\_frequency(cpu)})$$

对上面两个例子运用该公式，可以算出频率不变的任务利用率均为 25%。

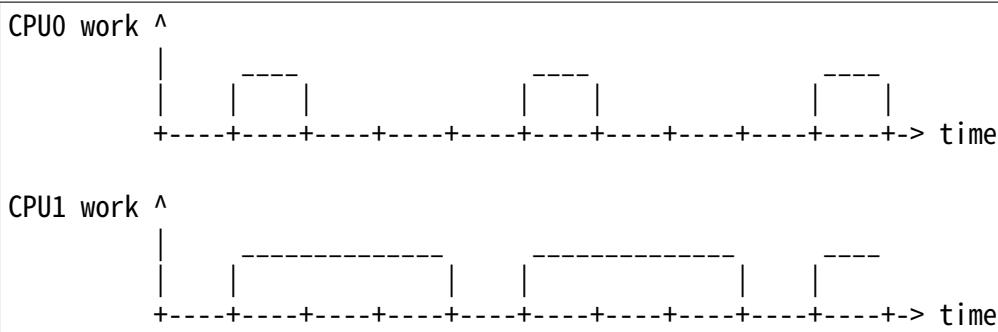
## 2.3 CPU 不变性

CPU 算力与任务利用率具有类型的效应，在算力不同的 CPU 上执行完全相同的工作负载，将算出不同的占用比。

考虑 1.3.2 节提到的系统，也就是说：

- $\text{capacity}(\text{CPU0}) = C$
- $\text{capacity}(\text{CPU1}) = C/3$

每个 CPU 按最大频率执行指定周期性工作负载，结果为：



也就是说，

- $\text{duty\_cycle}(p) == 25\%$ , 如果任务 p 在 CPU0 上按最大频率运行。
- $\text{duty\_cycle}(p) == 75\%$ , 如果任务 p 在 CPU1 上按最大频率运行。

任务利用率信号可按下面公式处理成 CPU 算力不变的：

$$\text{task\_util\_cpu\_inv}(p) = \text{duty\_cycle}(p) * (\text{capacity}(\text{cpu}) / \text{max\_capacity})$$

其中 `max_capacity` 是系统中最高的 CPU 算力。对上面的例子运用该公式，可以算出 CPU 算力不变的任务利用率均为 25%。

## 2.4 任务利用率不变量

频率和 CPU 算力不变性都需要被应用到任务利用率的计算中，以便求出真正的不变信号。任务利用率的伪计算公式是同时具备 CPU 和频率不变性的，也就是说，对于指定任务 p：

$$\text{task\_util\_inv}(p) = \text{duty\_cycle}(p) * \frac{\text{curr\_frequency}(\text{cpu})}{\text{max\_frequency}(\text{cpu})} * \frac{\text{capacity}(\text{cpu})}{\text{max\_capacity}}$$

也就是说，任务利用率不变量假定任务在系统中最高算力 CPU 上以最高频率运行，以此描述任务的行为。

在接下来的章节中提到的任何任务利用率，均是不变量的形式。

## 2.5 利用率估算

由于预测未来的水晶球不存在，当任务第一次变成可运行时，任务的行为和任务利用率均不能被准确预测。CFS 调度类基于实体负载跟踪机制（Per-Entity Load Tracking, PELT）维护了少量 CPU 和任务信号，其中之一可以算出平均利用率（与瞬时相反）。

这意味着，尽管运用“真实的”任务利用率（凭借水晶球）写出算力感知调度的准则，但是它的实现将只能用任务利用率的估算值。

## 3. 算力感知调度的需求

### 3.1 CPU 算力

当前，Linux 无法凭自身算出 CPU 算力，因此必须要有把这个信息传递给 Linux 的方式。每个架构必须为此定义 `arch_scale_cpu_capacity()` 函数。

arm 和 arm64 架构直接把这个信息映射到 `arch_topology` 驱动的 CPU scaling 数据中（译注：参考 `arch_topology.h` 的 `percpu` 变量 `cpu_scale`），它是从 `capacity-dmips-mhz` CPU binding 中衍生计算出来的。参见 `Documentation/devicetree/bindings/arm/cpu-capacity.txt`。

### 3.2 频率不变性

如 2.2 节所述，算力感知调度需要频率不变的任务利用率。每个架构必须为此定义 `arch_scale_freq_capacity(cpu)` 函数。

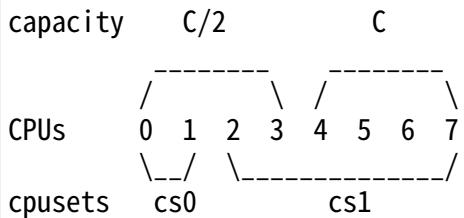
实现该函数要求计算出每个 CPU 当前以什么频率在运行。实现它的一种方式是利用硬件计数器（x86 的 APERF/MPERF，arm64 的 AMU），它能按 CPU 当前频率动态可扩展地升降递增计数器的速率。另一种方式是在 cpufreq 频率变化时直接使用钩子函数，内核此时感知到将要被切换的频率（也被 arm/arm64 实现了）。

## 4. 调度器拓扑结构

在构建调度域时，调度器将会发现系统是否表现为非对称 CPU 算力。如果是，那么：

- `sched_asym_cpucapacity` 静态键（static key）将使能。
- `SD_ASYM_CPU_CAPACITY_FULL` 标志位将在尽量最低调度域层级中被设置，同时要满足条件：调度域恰好完整包含某个 CPU 算力值的全部 CPU。
- `SD_ASYM_CPU_CAPACITY` 标志将在所有包含非对称 CPU 的调度域中被设置。

`sched_asym_cpucapacity` 静态键的设计意图是，保护为非对称 CPU 算力系统所准备的代码。不过要注意的是，这个键是系统范围可见的。想象下面使用了 `cpuset` 的步骤：



可以通过下面的方式创建：

```

mkdir /sys/fs/cgroup/cpuset/cs0
echo 0-1 > /sys/fs/cgroup/cpuset/cs0/cpuset.cpus
echo 0 > /sys/fs/cgroup/cpuset/cs0/cpuset.mems

mkdir /sys/fs/cgroup/cpuset/cs1
echo 2-7 > /sys/fs/cgroup/cpuset/cs1/cpuset.cpus
echo 0 > /sys/fs/cgroup/cpuset/cs1/cpuset.mems

echo 0 > /sys/fs/cgroup/cpuset/cpuset.sched_load_balance
  
```

由于“这是”非对称 CPU 算力系统，`sched_asym_cpucapacity` 静态键将使能。然而，CPU 0-1 对应的调度域层级，算力值仅有一个，该层级中 `SD_ASYM_CPUCAPACITY` 未被设置，它描述的是一个 SMP 区域，也应该被以此处理。

因此，“典型的”保护非对称 CPU 算力代码路径的代码模式是：

- 检查 `sched_asym_cpucapacity` 静态键
- 如果它被使能，接着检查调度域层级中 `SD_ASYM_CPUCAPACITY` 标志位是否出现

## 5. 算力感知调度的实现

### 5.1 CFS

#### 5.1.1 算力适应性 (fitness)

CFS 最主要的算力调度准则是：

```
task_util(p) < capacity(task_cpu(p))
```

它通常被称为算力适应性准则。也就是说，CFS 必须保证任务“适合”在某个 CPU 上运行。如果准则被违反，任务将要更长地消耗该 CPU，任务是 CPU 受限的 (CPU-bound)。

此外，`uclamp` 允许用户空间指定任务的最小和最大利用率，要么以 `sched_setattr()` 的方式，要么以 `cgroup` 接口的方式（参阅 `Documentation/admin-guide/cgroup-v2.rst`）。如其名字所暗示，`uclamp` 可以被用在前一条准则中限制 `task_util()`。

### 5.1.2 被唤醒任务的 CPU 选择

CFS 任务唤醒的 CPU 选择，遵循上面描述的算力适应性准则。在此之上，`uclamp` 被用来限制任务利用率，这令用户空间对 CFS 任务的 CPU 选择有更多的控制。也就是说，CFS 被唤醒任务的 CPU 选择，搜索满足以下条件的 CPU：

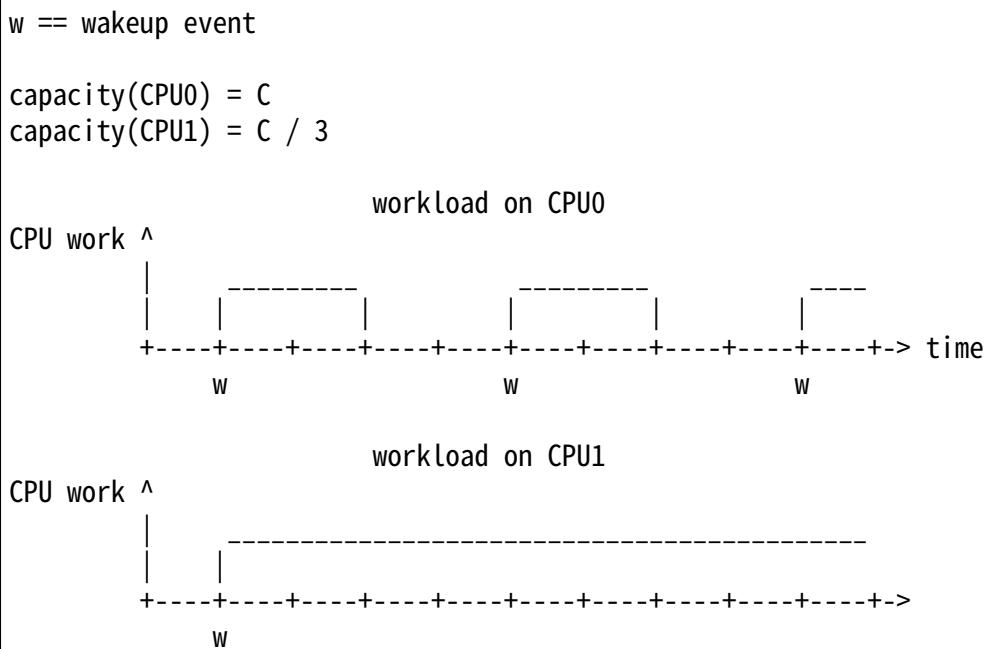
```
clamp(task_util(p), task_uclamp_min(p), task_uclamp_max(p)) < capacity(cpu)
```

通过使用 `uclamp`，举例来说，用户空间可以允许忙等待循环（100% 使用率）在任意 CPU 上运行，只要给它设置低的 `uclamp.max` 值。相反，`uclamp` 能强制一个周期性任务（比如，10% 利用率）在最高性能的 CPU 上运行，只要给它设置高的 `uclamp.min` 值。

**Note:** CFS 的被唤醒的任务的 CPU 选择，可被能耗感知调度（Energy Aware Scheduling, EAS）覆盖，在 `Documentation/scheduler/sched-energy.rst` 中描述。

### 5.1.3 负载均衡

被唤醒任务的 CPU 选择的一个病理性的例子是，任务几乎不睡眠，那么也几乎不发生唤醒。考虑：



该工作负载应该在 CPU0 上运行，不过如果任务满足以下条件之一：

- 一开始发生不合适的调度（不准确的初始利用率估计）
- 一开始调度正确，但突然需要更多的处理器功率

则任务可能变为 CPU 受限的，也就是说 `task_util(p) > capacity(task_cpu(p))`；CPU 算力调度准则被违反，将不会有任何唤醒事件来修复这个错误的 CPU 选择。

这种场景下的任务被称为“不合适的”(misfit)任务，处理这个场景的机制同样也以此命名。Misfit任务迁移借助CFS负载均衡器，更明确的说，是主动负载均衡的部分(用来迁移正在运行的任务)。当发生负载均衡时，如果一个misfit任务可以被迁移到一个相较当前运行的CPU具有更高算力的CPU上，那么misfit任务的主动负载均衡将被触发。

## 5.2 实时调度

### 5.2.1 被唤醒任务的CPU选择

实时任务唤醒时的CPU选择，搜索满足以下条件的CPU：

```
task_uclamp_min(p) <= capacity(task_cpu(cpu))
```

同时仍然允许接着使用常规的优先级限制。如果没有CPU能满足这个算力准则，那么将使用基于严格优先级的调度，CPU算力将被忽略。

## 5.3 最后期限调度

### 5.3.1 被唤醒任务的CPU选择

最后期限任务唤醒时的CPU选择，搜索满足以下条件的CPU：

```
task_bandwidth(p) < capacity(task_cpu(p))
```

同时仍然允许接着使用常规的带宽和截止期限限制。如果没有CPU能满足这个算力准则，那么任务依然在当前CPU队列中。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:<alexs@kernel.org>)。

---

**Original** Documentation/scheduler/sched-energy.rst

翻译 唐艺舟 Tang Yizhou <[tangyeechou@gmail.com](mailto:tangyeechou@gmail.com)>

## 能量感知调度

### 1. 简介

能量感知调度 (EAS) 使调度器有能力预测其决策对 CPU 所消耗的能量的影响。EAS 依靠一个能量模型 (EM) 来为每个任务选择一个节能的 CPU，同时最小化对吞吐率的影响。本文档致力于介绍 EAS 是如何工作的，它背后的主要设计决策是什么，以及使其运行所需的条件细节。

在进一步阅读之前，请注意，在撰写本文时：

/!\ EAS 不支持对称 CPU 拓扑的平台 /!\\

EAS 只在异构 CPU 拓扑结构（如 Arm 大小核，big.LITTLE）上运行。因为在这种情况下，通过调度来节约能量的潜力是最大的。

EAS 实际使用的 EM 不是由调度器维护的，而是一个专门的框架。关于这个框架的细节和它提供的内容，请参考其文档（见 Documentation/power/energy-model.rst）。

### 2. 背景和术语

从一开始就说清楚定义：

- 能量 = [焦耳] （比如供电设备上的电池提供的资源）
- 功率 = 能量/时间 = [焦耳/秒] = [瓦特]

EAS 的目标是最小化能量消耗，同时仍能将工作完成。也就是说，我们要最大化：

性能 [指令数/秒]

-----  
功率 [瓦特]

它等效于最小化：

能量 [焦耳]

-----  
指令数

同时仍然获得“良好”的性能。当前调度器只考虑性能目标，因此该式子本质上是一个可选的优化目标，它同时考虑了两个目标：能量效率和性能。

引入 EM 的想法是为了让调度器评估其决策的影响，而不是盲目地应用可能仅在部分平台有正面效果的节能技术。同时，EM 必须尽可能的简单，以最小化调度器的时延影响。

简而言之，EAS 改变了 CFS 任务分配给 CPU 的方式。当调度器决定一个任务应该在哪里运行时（在唤醒期间），EM 被用来在不损害系统吞吐率的情况下，从几个较好的候选 CPU 中挑选一个经预测能量消耗最优的 CPU。EAS 的预测依赖于对平台拓扑结构特定元素的了解，包括 CPU 的“算力”，以及它们各自的能量成本。

### 3. 拓扑信息

EAS（以及调度器的剩余部分）使用“算力”的概念来区分不同计算吞吐率的 CPU。一个 CPU 的“算力”代表了它在最高频率下运行时能完成的工作量，且这个值是相对系统中算力最大的 CPU 而言的。算力值被归一化为 1024 以内，并且可与由实体负载跟踪（PELT）机制算出的利用率信号做对比。由于有算力值和利用率值，EAS 能够估计一个任务/CPU 有多大/有多忙，并在评估性能与能量时将其考虑在内。CPU 算力由特定体系结构实现的 `arch_scale_cpu_capacity()` 回调函数提供。

EAS 使用的其余平台信息是直接从能量模型（EM）框架中读取的。一个平台的 EM 是一张表，表中每项代表系统中一个“性能域”的功率成本。（若要了解更多关于性能域的细节，见 `Documentation/power/energy-model.rst`）

当调度域被建立或重新建立时，调度器管理对拓扑代码中 EM 对象的引用。对于每个根域（rd），调度器维护一个与当前 `rd->span` 相交的所有性能域的单向链表。链表中的每个节点都包含一个指向 EM 框架所提供的结构体 `em_perf_domain` 的指针。

链表被附加在根域上，以应对独占的 cpuset 的配置。由于独占的 cpuset 的边界不一定与性能域的边界一致，不同根域的链表可能包含重复的元素。

**示例 1** 让我们考虑一个有 12 个 CPU 的平台，分成 3 个性能域，（pd0, pd4 和 pd8），按以下方式组织：

CPU:	0 1 2 3 4 5 6 7 8 9 10 11
PD:	--pd0-- --pd4-- ---pd8---
RD:	----rd1---- ----rd2----

现在，考虑用户空间决定将系统分成两个独占的 cpusets，因此创建了两个独立的根域，每个根域包含 6 个 CPU。这两个根域在上图中被表示为 rd1 和 rd2。由于 pd4 与 rd1 和 rd2 都有交集，它将同时出现于附加在这两个根域的“->pd”链表中：

- rd1->pd: pd0 -> pd4
- rd2->pd: pd4 -> pd8

请注意，调度器将为 pd4 创建两个重复的链表节点（每个链表中各一个）。然而这两个节点持有指向同一个 EM 框架的共享数据结构的指针。

由于对这些链表的访问可能与热插拔及其它事件并发生，因此它们受 RCU 锁保护，就像被调度器操控的拓扑结构体中剩下字段一样。

EAS 同样维护了一个静态键（`sched_energy_present`），当至少有一个根域满足 EAS 启动的所有条件时，这个键就会被启动。在第 6 节中总结了这些条件。

## 4. 能量感知任务放置

EAS 覆盖了 CFS 的任务唤醒平衡代码。在唤醒平衡时，它使用平台的 EM 和 PELT 信号来选择节能的目标 CPU。当 EAS 被启用时，`select_task_rq_fair()` 调用 `find_energy_efficient_cpu()` 来做任务放置决定。这个函数寻找在每个性能域中寻找具有最高剩余算力（CPU 算力 - CPU 利用率）的 CPU，因为它能让我们保持最低的频率。然后，该函数检查将任务放在新 CPU 相较依然放在之前活动的 `prev_cpu` 是否可以节省能量。

如果唤醒的任务被迁移，`find_energy_efficient_cpu()` 使用 `compute_energy()` 来估算系统将消耗多少能量。`compute_energy()` 检查各 CPU 当前的利用率情况，并尝试调整来“模拟”任务迁移。EM 框架提供了 API `em_pd_energy()` 计算每个性能域在给定的利用率条件下的预期能量消耗。

下面详细介绍一个优化能量消耗的任务放置决策的例子。

**示例 2** 让我们考虑一个有两个独立性能域的（伪）平台，每个性能域含有 2 个 CPU。CPU0 和 CPU1 是小核，CPU2 和 CPU3 是大核。

调度器必须决定将任务 P 放在哪个 CPU 上，这个任务的 `util_avg = 200`（平均利用率），`prev_cpu = 0`（上一次运行在 CPU0）。

目前 CPU 的利用率情况如下图所示。CPU 0-3 的 `util_avg` 分别为 400、100、600 和 500。每个性能域有三个操作性能值（OPP）。与每个 OPP 相关的 CPU 算力和功率成本列在能量模型表中。P 的 `util_avg` 在图中显示为“PP”：

CPU util.				Energy Model			
1024	- - - - -						
768	=====			Little   Big			
512	=====	- ##- - -		Cap   Pwr   Cap   Pwr			
341	-PP - - -	## ##		+-----+-----+-----+-----+			
170	-## - - -	## ##		170   50   512   400			
	## ##	## ##		341   150   768   800			
				512   300   1024   1700			
	CPU0	CPU1	CPU2	CPU3			
Current OPP: =====		Other OPP: - - -		util_avg (100 each): ##			

`find_energy_efficient_cpu()` 将首先在两个性能域中寻找具有最大剩余算力的 CPU。在这个例子中是 CPU1 和 CPU3。然后，它将估算，当 P 被放在它们中的任意一个时，系统的能耗，并检查这样做是否会比把 P 放在 CPU0 上节省一些能量。EAS 假定 OPPs 遵循利用率（这与 CPUFreq 监管器 `schedutil` 的行为一致。关于这个问题的更多细节，见第 6 节）。

**情况 1. P 被迁移到 CPU1:**

1024	- - - - -		
768	=====	Energy calculation:	
512	- # #- - - -	* CPU0: 200 / 341 * 150 = 88	
341	=====	* CPU1: 300 / 341 * 150 = 131	
170	- # # - - PP -	* CPU2: 600 / 768 * 800 = 625	
	## ##	* CPU3: 500 / 768 * 800 = 520	
	## ##	=> total_energy = 1364	
CPU0	CPU1	CPU2	CPU3

### 情况 2. P 被迁移到 CPU3:

1024	- - - - -		
768	=====	Energy calculation:	
512	- # #- - - -	* CPU0: 200 / 341 * 150 = 88	
341	=====	* CPU1: 100 / 341 * 150 = 43	
170	- # # - - -	* CPU2: 600 / 768 * 800 = 625	
	## ##	* CPU3: 700 / 768 * 800 = 729	
	## ##	=> total_energy = 1485	
CPU0	CPU1	CPU2	CPU3

### 情况 3. P 依旧留在 prev\_cpu/CPU0:

1024	- - - - -		
768	=====	Energy calculation:	
512	- # #- - - -	* CPU0: 400 / 512 * 300 = 234	
341	- PP - - - -	* CPU1: 100 / 512 * 300 = 58	
170	PP	* CPU2: 600 / 768 * 800 = 625	
	- # # - - -	* CPU3: 500 / 768 * 800 = 520	
	## ##	=> total_energy = 1437	
CPU0	CPU1	CPU2	CPU3

从这些计算结果来看，情况 1 的总能量最低。所以从节约能量的角度看，CPU1 是最佳候选者。

大核通常比小核更耗电，因此主要在任务不适合在小核运行时使用。然而，小核并不总是比大核节能。举例来说，对于某些系统，小核的高 OPP 可能比大核的低 OPP 能量消耗更高。因此，如果小核在某一特定时间点刚好有足够的利用率，在此刻被唤醒的小任务放在大核执行可能会更节能，尽管它在小核上运行也是合适的。

即使在大核所有 OPP 都不如小核 OPP 节能的情况下，在某些特定条件下，令小任务运行在大核上依然可能节能。事实上，将一个任务放在一个小核上可能导致整个性能域的 OPP 提高，这将增加已经在该性能域运行的任务的能量成本。如果唤醒的任务被放在一个大核上，它的执行成本可能比放在小核上更高，但这不会影响小核上的其它任务，这些任务将继续以较低的 OPP 运行。因此，当考虑 CPU 消耗的总能量时，在大核上运行一个任务的额外成本可能小于为所有其它运行在小核的任务提高 OPP 的成本。

上面的例子几乎不可能以一种通用的方式得到正确的结果；同时，对于所有平台，在不知道系统所有 CPU 每个不同 OPP 的运行成本时，也无法得到正确的结果。得益于基于 EM 的设计，EAS 应该能够正确处理这些问题而不会引发太多麻烦。然而，为了确保对高利用率场景的吞吐率造成的影响最小化，EAS 同样实现了另外一种叫“过度利用率”的机制。

## 5. 过度利用率

从一般的角度来看，EAS 能提供最大帮助的是那些涉及低、中 CPU 利用率的使用场景。每当 CPU 密集型的长任务运行，它们将需要所有的可用 CPU 算力，调度器将没有什么办法来节省能量同时又不损害吞吐率。为了避免 EAS 损害性能，一旦 CPU 被使用的算力超过 80%，它将被标记为“过度利用”。只要根域中没有 CPU 是过度利用状态，负载均衡被禁用，而 EAS 将覆盖唤醒平衡代码。EAS 很可能将负载放置在系统中能量效率最高的 CPU 而不是其它 CPU 上，只要不损害吞吐率。因此，负载均衡器被禁用以防止它打破 EAS 发现的节能任务放置。当系统未处于过度利用状态时，这样做是安全的，因为低于 80% 的临界点意味着：

- a. 所有的 CPU 都有一些空闲时间，所以 EAS 使用的利用率信号很可能准确地代表各种任务的“大小”。
- b. 所有任务，不管它们的 nice 值是多大，都应该被提供了足够多的 CPU 算力。
- c. 既然有多余的算力，那么所有的任务都必须定期阻塞/休眠，在唤醒时进行平衡就足够了。

只要一个 CPU 利用率超过 80% 的临界点，上述三个假设中至少有一个是不正确的。在这种情况下，整个根域的“过度利用”标志被设置，EAS 被禁用，负载均衡器被重新启用。通过这样做，调度器又回到了在 CPU 密集的条件下基于负载的算法做负载均衡。这更好地尊重了任务的 nice 值。

由于过度利用率的概念在很大程度上依赖于检测系统中是否有一些空闲时间，所以必须考虑（比 CFS）更高优先级的调度类（以及中断）“窃取”的 CPU 算力。像这样，对过度使用率的检测不仅要考虑 CFS 任务使用的算力，还需要考虑其它调度类和中断。

## 6. EAS 的依赖和要求

能量感知调度依赖系统的 CPU 具有特定的硬件属性，以及内核中的其它特性被启用。本节列出了这些依赖，并对如何满足这些依赖提供了提示。

## 6.1 - 非对称 CPU 拓扑

如简介所提，目前只有非对称 CPU 拓扑结构的平台支持 EAS。通过在运行时查询 SD\_ASYM\_CPUCAPACITY\_FULL 标志位是否在创建调度域时已设置来检查这一要求是否满足。

参阅 Documentation/scheduler/sched-capacity.rst 以了解在 sched\_domain 层次结构中设置此标志位所需满足的要求。

请注意，EAS 并非从根本上与 SMP 不兼容，但在 SMP 平台上还没有观察到明显的节能。这一限制可以在将来进行修改，如果被证明不是这样的话。

## 6.2 - 当前的能量模型

EAS 使用一个平台的 EM 来估算调度决策对能量的影响。因此，你的平台必须向 EM 框架提供能量成本表，以启动 EAS。要做到这一点，请参阅文档 Documentation/power/energy-model.rst 中的独立 EM 框架部分。

另请注意，调度域需要在 EM 注册后重建，以便启动 EAS。

EAS 使用 EM 对能量使用率进行预测决策，因此它在检查任务放置的可能选项时更加注重差异。对于 EAS 来说，EM 的功率值是以毫瓦还是以“抽象刻度”为单位表示并不重要。

## 6.3 - 能量模型复杂度

任务唤醒路径是时延敏感的。当一个平台的 EM 太复杂（太多 CPU，太多性能域，太多状态等），在唤醒路径中使用它的成本就会升高到不可接受。能量感知唤醒算法的复杂度为：

$$C = Nd * (Nc + Ns)$$

其中：Nd 是性能域的数量；Nc 是 CPU 的数量；Ns 是 OPP 的总数（例如：对于两个性能域，每个域有 4 个 OPP，则 Ns = 8）。

当调度域建立时，复杂性检查是在根域上进行的。如果一个根域的复杂度 C 恰好高于完全主观设定的 EM\_MAX\_COMPLEXITY 阈值（在本文写作时，是 2048），则 EAS 不会在此根域启动。

如果你的平台的能量模型的复杂度太高，EAS 无法在这个根域上使用，但你真的想用，那么你就只剩下两个可能的选择：

1. 将你的系统拆分成分离的、较小的、使用独占 cpuset 的根域，并在每个根域局部启用 EAS。这个方案的好处是开箱即用，但缺点是无法在根域之间实现负载均衡，这可能会导致总体系统负载不均衡。
2. 提交补丁以降低 EAS 唤醒算法的复杂度，从而使其能够在合理的时间内处理更大的 EM。

## 6.4 - Schedutil 监管器

EAS 试图预测 CPU 在不久的将来会在哪个 OPP 下运行，以估算它们的能量消耗。为了做到这一点，它假定 CPU 的 OPP 跟随 CPU 利用率变化而变化。

尽管在实践中很难对这一假设的准确性提供硬性保证（因为，举例来说硬件可能不会做它被要求做的事情），相对于其他 CPUFreq 监管器，schedutil 至少 \_ 请求 \_ 使用利用率信号计算的频率。因此，与 EAS 一起使用的唯一合理的监管器是 schedutil，因为它是唯一一个在频率请求和能量预测之间提供某种程度的一致性的监管器。

不支持将 EAS 与 schedutil 之外的任何其它监管器一起使用。

## 6.5 刻度不变性使用率信号

为了对不同的 CPU 和所有的性能状态做出准确的预测，EAS 需要频率不变的和 CPU 不变的 PELT 信号。这些信号可以通过每个体系结构定义的 arch\_scale{cpu,freq}\_capacity() 回调函数获取。

不支持在没有实现这两个回调函数的平台上使用 EAS。

## 6.6 多线程 (SMT)

当前实现的 EAS 是不感知 SMT 的，因此无法利用多线程硬件节约能量。EAS 认为线程是独立的 CPU，这实际上对性能和能量消耗都是不利的。

不支持在 SMT 上使用 EAS。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/scheduler/schedutil.rst

翻译 唐艺舟 Tang Yizhou <[tangyeechou@gmail.com](mailto:tangyeechou@gmail.com)>

## Schedutil

**Note:** 本文所有内容都假设频率和工作算力之间存在线性关系。我们知道这是有瑕疵的，但这是最可行的近似处理。

### PELT (实体负载跟踪, Per Entity Load Tracking)

通过 PELT，我们跟踪了各种调度器实体的一些指标，从单个任务到任务组分片到 CPU 运行队列。我们使用指数加权移动平均数 (Exponentially Weighted Moving Average, EWMA) 作为其基础，每个周期 (1024us) 都会衰减，衰减速率满足  $y^{32} = 0.5$ 。也就是说，最近的 32ms 贡献负载的一半，而历史上的其它时间则贡献另一半。

具体而言：

$$\begin{aligned}\text{ewma\_sum}(u) &:= u_0 + u_1 * y + u_2 * y^2 + \dots \\ \text{ewma}(u) &= \text{ewma\_sum}(u) / \text{ewma\_sum}(1)\end{aligned}$$

由于这本质上是一个无限几何级数的累加，结果是可组合的，即  $\text{ewma}(A) + \text{ewma}(B) = \text{ewma}(A+B)$ 。这个属性是关键，因为它提供了在任务迁移时重新组合平均数的能力。

请注意，阻塞态的任务仍然对累加值（任务组分片和 CPU 运行队列）有贡献，这反映了它们在恢复运行后的预期贡献。

利用这一点，我们跟踪 2 个关键指标：“运行”和“可运行”。“运行”反映了一个调度实体在 CPU 上花费的时间，而“可运行”反映了一个调度实体在运行队列中花费的时间。当只有一个任务时，这两个指标是相同的，但一旦出现对 CPU 的争用，“运行”将减少以反映每个任务在 CPU 上花费的时间，而“可运行”将增加以反映争用的激烈程度。

更多细节见：kernel/sched/pelt.c

### 频率 / CPU 不变性

因为 CPU 频率在 1GHz 时利用率为 50% 和 CPU 频率在 2GHz 时利用率为 50% 是不一样的，同样在小核上运行时利用率为 50% 和在大核上运行时利用率为 50% 是不一样的，我们允许架构以两个比率来伸缩时间差，其中一个是动态电压频率升降 (Dynamic Voltage and Frequency Scaling, DVFS) 比率，另一个是微架构比率。

对于简单的 DVFS 架构（软件有完全控制能力），我们可以很容易地计算该比率：

```
f_cur
r_dvfs := -----
          f_max
```

对于由硬件控制 DVFS 的更多动态系统，我们使用硬件计数器（Intel APERF/MPERF，ARMv8.4-AMU）来计算这一比率。具体到 Intel，我们使用：

```

APERF
f_cur := ----- * P0
      MPERF

        4C-turbo; 如果可用并且使能了 turbo
f_max := { 1C-turbo; 如果使能了 turbo
          P0;       其它情况

r_dvfs := min( 1, ----- )
            f_cur
            f_max

```

我们选择 4C turbo 而不是 1C turbo，以使其更持久性略微更强。

$r_{cpu}$  被定义为当前 CPU 的最高性能水平与系统中任何其它 CPU 的最高性能水平的比率。

$$r_{tot} = r_{dvfs} * r_{cpu}$$

其结果是，上述“运行”和“可运行”的指标变成 DVFS 无关和 CPU 型号无关了。也就是说，我们可以在 CPU 之间转移和比较它们。

更多细节见：

- kernel/sched/pelt.h:update\_rq\_clock\_pelt()
- arch/x86/kernel/smpboot.c:”APERF/MPERF frequency ratio computation.”
- 算力感知调度:” 1. CPU Capacity + 2. Task utilization”

## UTIL\_EST / UTIL\_EST\_FASTUP

由于周期性任务的平均数在睡眠时会衰减，而在运行时其预期利用率会和睡眠前相同，因此它们在再次运行后会面临（DVFS）的上涨。

为了缓解这个问题，（一个默认使能的编译选项）UTIL\_EST 驱动一个无限脉冲响应（Infinite Impulse Response, IIR）的 EWMA，“运行”值在出队时是最高的。另一个默认使能的编译选项 UTIL\_EST\_FASTUP 修改了 IIR 滤波器，使其允许立即增加，仅在利用率下降时衰减。

进一步，运行队列的（可运行任务的）利用率之和由下式计算：

$$util\_est := \text{Sum}_t \max(t_{running}, t_{util\_est\_ewma})$$

更多细节见：kernel/sched/fair.c:util\_est\_dequeue()

## UCLAMP

可以在每个 CFS 或 RT 任务上设置有效的 u\_min 和 u\_max clamp 值（译注：clamp 可以理解为类似滤波器的能力，它定义了有效取值范围的最大值和最小值）；运行队列为所有正在运行的任务保持这些 clamp 的最大聚合值。

更多细节见：include/uapi/linux/sched/types.h

## Schedutil / DVFS

每当调度器的负载跟踪被更新时（任务唤醒、任务迁移、时间流逝），我们都会调用 schedutil 来更新硬件 DVFS 状态。

其基础是 CPU 运行队列的“运行”指标，根据上面的内容，它是 CPU 的频率不变的利用率估计值。由此我们计算出一个期望的频率，如下：

```

        max( running, util_est ); 如果使能 UTIL_EST
u_cfs := { running;           其它情况

        clamp( u_cfs + u_rt, u_min, u_max ); 如果使能 UCLAMP_TASK
u_clamp := { u_cfs + u_rt;     其它情况

u := u_clamp + u_irq + u_dl;   [估计值。更多细节见源代码]

f_des := min( f_max, 1.25 u * f_max )

```

关于 IO-wait 的说明：当发生更新是因为任务从 IO 完成中唤醒时，我们提升上面的“u”。

然后，这个频率被用来选择一个 P-state 或 OPP，或者直接混入一个发给硬件的 CPPC 式请求。

关于截止期限调度器的说明：截止期限任务（偶发任务模型）使我们能够计算出满足工作负荷所需的硬 f\_min 值。

因为这些回调函数是直接来自调度器的，所以 DVFS 的硬件交互应该是“快速”和非阻塞的。在硬件交互缓慢和昂贵的时候，schedutil 支持 DVFS 请求限速，不过会降低效率。

更多信息见：kernel/sched/cpufreq\_schedutil.c

## 注意

- 在低负载场景下，DVFS 是最相关的，“运行”的值将密切反映利用率。
- 在负载饱和的场景下，任务迁移会导致一些瞬时性的使用率下降。假设我们有一个 CPU，有 4 个任务占用导致其饱和，接下来我们将一个任务迁移到另一个空闲 CPU 上，旧的 CPU 的“运行”值将为 0.75，而新的 CPU 将获得 0.25。这是不可避免的，而且随着时间流逝将自动修正。另注，由于没有空闲时间，我们还能保证 f\_max 值吗？

- 上述大部分内容是关于避免 DVFS 下滑，以及独立的 DVFS 域发生负载迁移时不得不重新学习/提升频率。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/scheduler/sched-nice-design.rst

翻译 唐艺舟 Tang Yizhou <[tangyeechou@gmail.com](mailto:tangyeechou@gmail.com)>

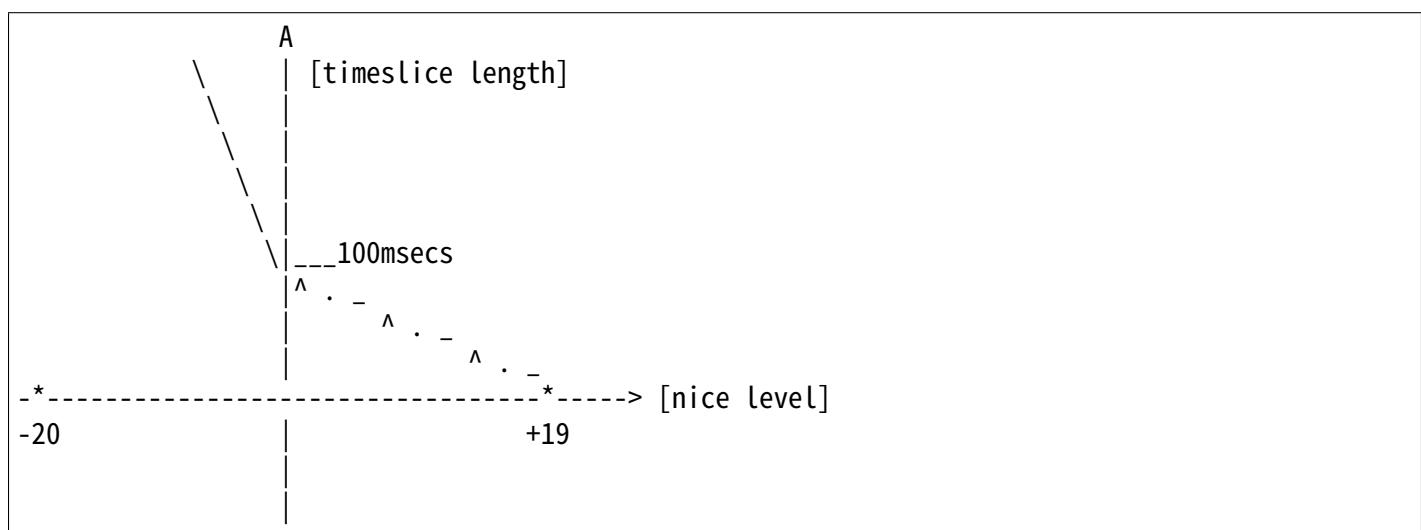
## 调度器 nice 值设计

本文档解释了新的 Linux 调度器中修改和精简后的 nice 级别的实现思路。

Linux 的 nice 级别总是非常脆弱，人们持续不断地缠着我们，让 nice +19 的任务占用更少的 CPU 时间。

不幸的是，在旧的调度器中，这不是那么容易实现的（否则我们早就做到了），因为对 nice 级别的支持在历史上是与时间片长度耦合的，而时间片单位是由 HZ 滴答驱动的，所以最小的时间片是 1/HZ。

在 O(1) 调度器中（2003 年），我们改变了负的 nice 级别，使它们比 2.4 内核更强（人们对这一变化很满意），而且我们还故意校正了线性时间片准则，使得 nice +19 的级别 正好 是 1 jiffy。为了让大家更好地理解它，时间片的图会是这样的（质量不佳的 ASCII 艺术提醒！）：



因此，如果有人真的想 renice 任务，相较线性规则，+19 会给出更大的效果（改变 ABI 来扩展优先级的解决方案在早期就被放弃了）。

这种方法在一定程度上奏效了一段时间，但后来 HZ=1000 时，它导致 1 jiffy 为 1ms，这意味着 0.1% 的

CPU 使用率，我们认为这有点过度。过度 \_ 不是 \_ 因为它表示的 CPU 使用率过小，而是因为它引发了过于频繁（每毫秒 1 次）的重新调度（因此会破坏缓存，等等。请记住，硬件更弱、cache 更小是很久以前的事了，当时人们在 nice +19 级别运行数量颇多的应用程序）。

因此，对于 HZ=1000，我们将 nice +19 改为 5 毫秒，因为这感觉像是正确的最小粒度——这相当于 5% 的 CPU 利用率。但 nice +19 的根本的 HZ 敏感属性依然保持不变，我们没有收到过关于 nice +19 在 CPU 利用率方面太 \_ 弱 \_ 的任何抱怨，我们只收到过它（依然）太 \_ 强 \_ 的抱怨:-)。

总结一下：我们一直想让 nice 各级别一致性更强，但在 HZ 和 jiffies 的限制下，以及 nice 级别与时间片、调度粒度耦合是令人讨厌的设计，这一目标并不真正可行。

第二个关于 Linux nice 级别支持的抱怨是（不那么频繁，但仍然定期发生），它在原点周围的不对称性（你可以在上面的图片中看到），或者更准确地说：事实上 nice 级别的行为取决于 \_ 绝对的 \_ nice 级别，而 nice 应用程序接口本身从根本上说是“相对”的：

```
int nice(int inc);
asmlinkage long sys_nice(int increment)
```

（第一个是 glibc 的应用程序接口，第二个是 syscall 的应用程序接口）注意，“inc”是相对当前 nice 级别而言的，类似 bash 的“nice”命令等工具是这个相对性应用程序接口的镜像。

在旧的调度器中，举例来说，如果你以 nice +1 启动一个任务，并以 nice +2 启动另一个任务，这两个任务的 CPU 分配将取决于父外壳程序的 nice 级别——如果它是 nice -10，那么 CPU 的分配不同于 +5 或 +10。

第三个关于 Linux nice 级别支持的抱怨是，负数 nice 级别“不够有力”，以很多人不得不诉诸于实时调度优先级来运行音频（和其它多媒体）应用程序，比如 SCHED\_FIFO。但这也造成了其它问题：SCHED\_FIFO 未被证明是免于饥饿的，一个有问题的 SCHED\_FIFO 应用程序也会锁住运行良好的系统。

v2.6.23 版内核的新调度器解决了这三种类型的抱怨：

为了解决第一个抱怨（nice 级别不够“有力”），调度器与“时间片”、HZ 的概念解耦（调度粒度被处理成一个和 nice 级别独立的概念），因此有可能实现更好、更一致的 nice +19 支持：在新的调度器中，nice +19 的任务得到一个 HZ 无关的 1.5%CPU 使用率，而不是旧版调度器中 3%-5%-9% 的可变范围。

为了解决第二个抱怨（nice 各级别不一致），新调度器令调用 nice(1) 对各任务的 CPU 利用率有相同的影响，无论其绝对 nice 级别如何。所以在新调度器上，运行一个 nice +10 和一个 nice +11 的任务会与运行一个 nice -5 和一个 nice -4 的任务的 CPU 利用率分割是相同的（一个会得到 55% 的 CPU，另一个会得到 45%）。这是为什么 nice 级别被改为“乘法”（或指数）——这样的话，不管你从哪个级别开始，“相对”结果将总是一样的。

第三个抱怨（负数 nice 级别不够“有力”，并迫使音频应用程序在更危险的 SCHED\_FIFO 调度策略下运行）几乎被新的调度器自动解决了：更强的负数级别具有重新校正 nice 级别动态范围的自动化副作用。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题, 请联系该文件的译者, 或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

### Original Documentation/scheduler/sched-stats.rst

翻译 唐艺舟 Tang Yizhou <[tangyeechou@gmail.com](mailto:tangyeechou@gmail.com)>

## 调度器统计数据

第 15 版 schedstats 去掉了 sched\_yield 的一些计数器: yld\_exp\_empty, yld\_act\_empty 和 yld\_both\_empty。在其它方面和第 14 版完全相同。

第 14 版 schedstats 包括对 sched\_domains (译注: 调度域) 的支持, 该特性进入内核主线 2.6.20, 不过这一版 schedstats 与 2.6.13-2.6.19 内核的版本 12 的统计数据是完全相同的 (内核未发布第 13 版)。有些计数器按每个运行队列统计是更有意义的, 其它则按每个调度域统计是更有意义的。注意, 调度域 (以及它们的附属信息) 仅在开启 CONFIG\_SMP 的机器上是相关的和可用的。

在第 14 版 schedstat 中, 每个被列出的 CPU 至少会有一级域统计数据, 且很可能有一个以上的域。在这个实现中, 域没有特别的名字, 但是编号最高的域通常在机器上所有的 CPU 上仲裁平衡, 而 domain0 是最紧密聚焦的域, 有时仅在一对 CPU 之间进行平衡。此时, 没有任何体系结构需要 3 层以上的域。域统计数据中的第一个字段是一个位图, 表明哪些 CPU 受该域的影响。

这些字段是计数器, 而且只能递增。使用这些字段的程序将需要从基线观测开始, 然后在后续每一个观测中计算出计数器的变化。一个能以这种方式处理其中很多字段的 perl 脚本可见

<http://eaglet.pdxhosts.com/rick/linux/schedstat/>

请注意, 任何这样的脚本都必须是特定于版本的, 改变版本的主要原因是输出格式的变化。对于那些希望编写自己的脚本的人, 可以参考这里描述的各个字段。

## CPU 统计数据

cpu<N> 1 2 3 4 5 6 7 8 9

第一个字段是 sched\_yield() 的统计数据:

1) sched\_yield() 被调用了 # 次

接下来的三个是 schedule() 的统计数据:

2) 这个字段是一个过时的数组过期计数, 在 O(1) 调度器中使用。为了 ABI 兼容性, 我们保留了它, 但它总是被设置为 0。

3) schedule() 被调用了 # 次

4) 调用 schedule() 导致处理器变为空闲了 # 次

接下来的两个是 try\_to\_wake\_up() 的统计数据:

- 5) try\_to\_wake\_up() 被调用了 # 次
- 6) 调用 try\_to\_wake\_up() 导致本地 CPU 被唤醒了 # 次

接下来的三个统计数据描述了调度延迟：

- 7) 本处理器运行任务的总时间，单位是 jiffies
- 8) 本处理器任务等待运行的时间，单位是 jiffies
- 9) 本 CPU 运行了 # 个时间片

## 域统计数据

对于每个被描述的 CPU，和它相关的每一个调度域均会产生下面一行数据（注意，如果 CONFIG\_SMP 没有被定义，那么 \* 没有 \* 调度域被使用，这些行不会出现在输出中）。

```
domain<N> <cpumask> 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26  
27 28 29 30 31 32 33 34 35 36
```

第一个字段是一个位掩码，表明该域在操作哪些 CPU。

接下来的 24 个字段是 load\_balance() 函数的各个统计数据，按空闲类型分组（空闲，繁忙，新空闲）：

- 1) 当 CPU 空闲时，load\_balance() 在这个调度域中被调用了 # 次
- 2) 当 CPU 空闲时，load\_balance() 在这个调度域中被调用，但是发现负载无需均衡 # 次
- 3) 当 CPU 空闲时，load\_balance() 在这个调度域中被调用，试图迁移 1 个或更多任务且失败了 # 次
- 4) 当 CPU 空闲时，load\_balance() 在这个调度域中被调用，发现不均衡（如果有） # 次
- 5) 当 CPU 空闲时，pull\_task() 在这个调度域中被调用 # 次
- 6) 当 CPU 空闲时，尽管目标任务是热缓存状态，pull\_task() 依然被调用 # 次
- 7) 当 CPU 空闲时，load\_balance() 在这个调度域中被调用，未能找到更繁忙的队列 # 次
- 8) 当 CPU 空闲时，在调度域中找到了更繁忙的队列，但未找到更繁忙的调度组 # 次
- 9) 当 CPU 繁忙时，load\_balance() 在这个调度域中被调用了 # 次
- 10) 当 CPU 繁忙时，load\_balance() 在这个调度域中被调用，但是发现负载无需均衡 # 次
- 11) 当 CPU 繁忙时，load\_balance() 在这个调度域中被调用，试图迁移 1 个或更多任务且失败了 # 次
- 12) 当 CPU 繁忙时，load\_balance() 在这个调度域中被调用，发现不均衡（如果有） # 次
- 13) 当 CPU 繁忙时，pull\_task() 在这个调度域中被调用 # 次
- 14) 当 CPU 繁忙时，尽管目标任务是热缓存状态，pull\_task() 依然被调用 # 次
- 15) 当 CPU 繁忙时，load\_balance() 在这个调度域中被调用，未能找到更繁忙的队列 # 次
- 16) 当 CPU 繁忙时，在调度域中找到了更繁忙的队列，但未找到更繁忙的调度组 # 次
- 17) 当 CPU 新空闲时，load\_balance() 在这个调度域中被调用了 # 次

- 18) 当 CPU 新空闲时, `load_balance()` 在这个调度域中被调用, 但是发现负载无需均衡 # 次
- 19) 当 CPU 新空闲时, `load_balance()` 在这个调度域中被调用, 试图迁移 1 个或更多任务且失败了 # 次
- 20) 当 CPU 新空闲时, `load_balance()` 在这个调度域中被调用, 发现不均衡 (如果有) # 次
- 21) 当 CPU 新空闲时, `pull_task()` 在这个调度域中被调用 # 次
- 22) 当 CPU 新空闲时, 尽管目标任务是热缓存状态, `pull_task()` 依然被调用 # 次
- 23) 当 CPU 新空闲时, `load_balance()` 在这个调度域中被调用, 未能找到更繁忙的队列 # 次
- 24) 当 CPU 新空闲时, 在调度域中找到了更繁忙的队列, 但未找到更繁忙的调度组 # 次

接下来的 3 个字段是 `active_load_balance()` 函数的各个统计数据:

- 25) `active_load_balance()` 被调用了 # 次
- 26) `active_load_balance()` 被调用, 试图迁移 1 个或更多任务且失败了 # 次
- 27) `active_load_balance()` 被调用, 成功迁移了 # 次任务

接下来的 3 个字段是 `sched_balance_exec()` 函数的各个统计数据:

- 28) `sbe_cnt` 不再被使用
- 29) `sbe_balanced` 不再被使用
- 30) `sbe_pushed` 不再被使用

接下来的 3 个字段是 `sched_balance_fork()` 函数的各个统计数据:

- 31) `sbf_cnt` 不再被使用
- 32) `sbf_balanced` 不再被使用
- 33) `sbf_pushed` 不再被使用

接下来的 3 个字段是 `try_to_wake_up()` 函数的各个统计数据:

- 34) 在这个调度域中调用 `try_to_wake_up()` 唤醒任务时, 任务在调度域中一个和上次运行不同的新 CPU 上运行了 # 次
- 35) 在这个调度域中调用 `try_to_wake_up()` 唤醒任务时, 任务被迁移到发生唤醒的 CPU 次数为 #, 因为该任务在原 CPU 是冷缓存状态
- 36) 在这个调度域中调用 `try_to_wake_up()` 唤醒任务时, 引发被动负载均衡 # 次

### /proc/<pid>/schedstat

schedstats 还添加了一个新的/proc/<pid>/schedstat 文件，来提供一些进程级的相同信息。这个文件中，有三个字段与该进程相关：

- 1) 在 CPU 上运行花费的时间
- 2) 在运行队列上等待的时间
- 3) 在 CPU 上运行了 # 个时间片

可以很容易地编写一个程序，利用这些额外的字段来报告一个特定的进程或一组进程在调度器策略下的表现如何。这样的程序的一个简单版本可在下面的链接找到

<http://eaglet.pdxhosts.com/rick/linux/schedstat/v12/latency.c>

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

**Original** Documentation/scheduler/sched-debug.rst

翻译 唐艺舟 Tang Yizhou <[tangyeechou@gmail.com](mailto:tangyeechou@gmail.com)>

### 调度器 debugfs

用配置项 CONFIG\_SCHED\_DEBUG=y 启动内核后，将可以访问/sys/kernel/debug/sched 下的调度器专用调试文件。其中一些文件描述如下。

### numa\_balancing

*numa\_balancing* 目录用来存放控制非统一内存访问 (NUMA) 平衡特性的相关文件。如果该特性导致系统负载太高，那么可以通过 *scan\_period\_min\_ms*, *scan\_delay\_ms*, *scan\_period\_max\_ms*, *scan\_size\_mb* 文件控制 NUMA 缺页的内核采样速率。

## **scan\_period\_min\_ms, scan\_delay\_ms, scan\_period\_max\_ms, scan\_size\_mb**

自动 NUMA 平衡会扫描任务地址空间，检测页面是否被正确放置，或者数据是否应该被迁移到任务正在运行的本地内存结点，此时需解映射页面。每个“扫描延迟”(scan delay)时间之后，任务扫描其地址空间中下一批“扫描大小”(scan size)个页面。若抵达内存地址空间末尾，扫描器将从头开始重新扫描。

结合来看，“扫描延迟”和“扫描大小”决定扫描速率。当“扫描延迟”减小时，扫描速率增加。“扫描延迟”和每个任务的扫描速率都是自适应的，且依赖历史行为。如果页面被正确放置，那么扫描延迟就会增加；否则扫描延迟就会减少。“扫描大小”不是自适应的，“扫描大小”越大，扫描速率越高。

更高的扫描速率会产生更高的系统开销，因为必须捕获缺页异常，并且潜在地必须迁移数据。然而，当扫描速率越高，若工作负载模式发生变化，任务的内存将越快地迁移到本地结点，由于远程内存访问而产生的性能影响将降到最低。下面这些文件控制扫描延迟的阈值和被扫描的页面数量。

`scan_period_min_ms` 是扫描一个任务虚拟内存的最长时间，单位是毫秒。它有效地控制了每个任务的最大扫描速率。

`scan_delay_ms` 是一个任务初始化创建 (fork) 时，第一次使用的“扫描延迟”。

`scan_period_max_ms` 是扫描一个任务虚拟内存的最大时间，单位是毫秒。它有效地控制了每个任务的最小扫描速率。

`scan_size_mb` 是一次特定的扫描中，要扫描多少兆字节 (MB) 对应的页面数。

TODOList:

sched-deadline sched-rt-group

text\_files

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/vm/index.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译

### \* Linux 内存管理文档

这是一个关于 Linux 内存管理 (mm) 子系统内部的文档集，其中有不同层次的细节，包括注释和邮件列表的回复，用于阐述数据结构和算法的基本情况。如果你正在寻找关于简单分配内存的建议，请参阅 ([内存分配指南](#))。对于控制和调整指南，请参阅 ([Documentation/admin-guide/mm/index](#))。TODO: 待引用文档集被翻译完毕后请及时修改此处)

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/vm/active\_mm.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译

### Active MM

这是一封 linux 之父回复开发者的一封邮件，所以翻译时我尽量保持邮件格式的完整。

List: linux-kernel  
Subject: Re: active\_mm  
From: Linus Torvalds <torvalds () transmeta ! com>  
Date: 1999-07-30 21:36:24

因为我并不经常写解释，所以已经抄送到 linux-kernel 邮件列表，而当我做这些，且更多的人在阅读它们时，我觉得棒极了。

1999 年 7 月 30 日 星期五， David Mosberger 写道：

>  
> 是否有一个简短的描述，说明 task\_struct 中的  
> "mm" 和 "active\_mm" 应该如何使用？（如果  
> 这个问题在邮件列表中讨论过，我表示歉意--我刚  
> 刚度假回来，有一段时间没能关注 linux-kernel 了）。

基本上，新的设定是：

- 我们有“真实地址空间”和“匿名地址空间”。区别在于，匿名地址空间根本不关心用户级页表，所以当我们做上下文切换到匿名地址空间时，我们只是让以前的地址空间处于活动状态。

一个“匿名地址空间”的明显用途是任何不需要任何用户映射的线程--所有的内核线

程基本上都属于这一类，但即使是“真正的”线程也可以暂时说在一定时间内它们不会对用户空间感兴趣，调度器不妨试着避免在切换 VM 状态上浪费时间。目前只有老式的 `bdflush sync` 能做到这一点。

- “`tsk->mm`” 指向“真实地址空间”。对于一个匿名进程来说，`tsk->mm` 将是 `NULL`，其逻辑原因是匿名进程实际上根本就“没有”真正的地址空间。
- 然而，我们显然需要跟踪我们为这样的匿名用户“偷用”了哪个地址空间。为此，我们有“`tsk->active_mm`”，它显示了当前活动的地址空间是什么。

规则是，对于一个有真实地址空间的进程（即 `tsk->mm` 是 `non-NULL`），`active_mm` 显然必须与真实的 `mm` 相同。

对于一个匿名进程，`tsk->mm == NULL`，而 `tsk->active_mm` 是匿名进程运行时“借用”的 `mm`。当匿名进程被调度走时，借用的地址空间被返回并清除。

为了支持所有这些，“`struct mm_struct`”现在有两个计数器：一个是“`mm_users`”计数器，即有多少“真正的地址空间用户”，另一个是“`mm_count`”计数器，即“lazy”用户（即匿名用户）的数量，如果有任何真正的用户，则加 1。

通常情况下，至少有一个真正的用户，但也可能是真正的用户在另一个 CPU 上退出，而一个 `lazy` 的用户仍在活动，所以你实际上得到的情况是，你有一个地址空间 \*\* 只 \*\* 被 `lazy` 的用户使用。这通常是一个短暂的生命周期状态，因为一旦这个线程被安排给一个真正的线程，这个“僵尸”`mm`就会被释放，因为“`mm_count`”变成了零。

另外，一个新的规则是，\*\* 没有人 \*\* 再把“`init_mm`”作为一个真正的 MM 了。“`init_mm`”应该被认为只是一个“没有其他上下文时的 `lazy` 上下文”，事实上，它主要是在启动时使用，当时还没有真正的 VM 被创建。因此，用来检查的代码

```
if (current->mm == &init_mm)
```

一般来说，应该用

```
if (!current->mm)
```

取代上面的写法（这更有意义--测试基本上是“我们是否有一个用户环境”，并且通常由缺页异常处理程序和类似的东西来完成）。

总之，我刚才在 `ftp.kernel.org` 上放了一个 `pre-patch-2.3.13-1`，因为它稍微改变了接口以适配 `alpha`（谁会想到呢，但 `alpha` 体系结构上下文切换代码实际上最终是最丑陋的之一--不像其他架构的 MM 和寄存器状态是分开的，`alpha` 的 PALcode 将两者连接起来，你需要同时切换两者）。

（文档来源 <http://marc.info/?l=linux-kernel&m=93337278602211&w=2>）

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题, 请联系该文件的译者, 或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

### Original Documentation/vm/balance.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译

## 内存平衡

2000 年 1 月开始, 作者: Kanoj Sarcar <[kanoj@sgi.com](mailto:kanoj@sgi.com)>

对于! \_\_GFP\_HIGH 和! \_\_GFP\_KSWAPD\_RECLAIM 以及非 \_\_GFP\_IO 的分配, 需要进行内存平衡。

调用者避免回收的第一个原因是调用者由于持有自旋锁或处于中断环境中而无法睡眠。第二个原因可能是, 调用者愿意在不产生页面回收开销的情况下分配失败。这可能发生在有 0 阶回退选项的机会主义高阶分配请求中。在这种情况下, 调用者可能也希望避免唤醒 kswapd。

\_\_GFP\_IO 分配请求是为了防止文件系统死锁。

在没有非睡眠分配请求的情况下, 做平衡似乎是有害的。页面回收可以被懒散地启动, 也就是说, 只有在需要的时候 (也就是区域的空闲内存为 0), 而不是让它成为一个主动的过程。

也就是说, 内核应该尝试从直接映射池中满足对直接映射页的请求, 而不是回退到 dma 池中, 这样就可以保持 dma 池为 dma 请求 (不管是原子的) 所填充。类似的争论也适用于高内存和直接映射的页面。相反, 如果有很多空闲的 dma 页, 最好是通过从 dma 池中分配一个来满足常规的内存请求, 而不是产生常规区域平衡的开销。

在 2.2 中, 只有当空闲页总数低于总内存的 1/64 时, 才会启动内存平衡/页面回收。如果 dma 和常规内存的比例合适, 即使 dma 区完全空了, 也很可能不会进行平衡。2.2 已经在不同内存大小的生产机器上运行, 即使有这个问题存在, 似乎也做得不错。在 2.3 中, 由于 HIGHMEM 的存在, 这个问题变得更加严重。

在 2.3 中, 区域平衡可以用两种方式之一来完成: 根据区域的大小 (可能是低级区域的大小), 我们可以在初始化阶段决定在平衡任何区域时应该争取多少空闲页。好的方面是, 在平衡的时候, 我们不需要看低级区的大小, 坏的方面是, 我们可能会因为忽略低级区可能较低的使用率而做过于频繁的平衡。另外, 只要对分配程序稍作修改, 就有可能将 memclass() 宏简化为一个简单的等式。

另一个可能的解决方案是, 我们只在一个区 和其所有低级区的空闲内存低于该区及其低级区总内存的 1/64 时进行平衡。这就解决了 2.2 的平衡问题, 并尽可能地保持了与 2.2 行为的接近。另外, 平衡算法在各种架构上的工作方式也是一样的, 这些架构有不同数量和类型的内存区。如果我们想变得更花哨一点, 我们可以在未来为不同区域的自由页面分配不同的权重。

请注意, 如果普通区的大小与 dma 区相比是巨大的, 那么在决定是否平衡普通区的时候, 考虑空闲的 dma 页就变得不那么重要了。那么第一个解决方案就变得更有吸引力。

所附的补丁实现了第二个解决方案。它还“修复”了两个问题: 首先, 在低内存条件下, kswapd 被唤醒, 就像 2.2 中的非睡眠分配。第二, HIGHMEM 区也被平衡了, 以便给 replace\_with\_highmem() 一个争取获

得 HIGHMEM 页的机会，同时确保 HIGHMEM 分配不会落回普通区。这也确保了 HIGHMEM 页不会被泄露（例如，在一个 HIGHMEM 页在交换缓存中但没有被任何人使用的情况下）。

kswapd 还需要知道它应该平衡哪些区。kswapd 主要在无法进行平衡的情况下需要的，可能是因为所有的分配请求都来自中断上下文，而所有的进程上下文都在睡眠。对于 2.3，kswapd 并不真正需要平衡高内存区，因为中断上下文并不请求高内存页。kswapd 看 zone 结构体中的 zone\_wake\_kswapd 字段来决定一个区是否需要平衡。

如果从进程内存和 shm 中偷取页面可以减轻该页面节点中任何区的内存压力，而该区的内存压力已经低于其水位，则会进行偷取。

`watermark[WMARK_MIN/WMARK_LOW/WMARK_HIGH]/low_on_memory/zone_wake_kswapd`:  
这些是每个区的字段，用于确定一个区何时需要平衡。当页面数低于水位 [WMARK\_MIN] 时，`hysteric` 的字段 `low_on_memory` 被设置。这个字段会一直被设置，直到空闲页数变成水位 [WMARK\_HIGH]。当 `low_on_memory` 被设置时，页面分配请求将尝试释放该区域的一些页面（如果请求中设置了 GFP\_WAIT）。与此相反的是，决定唤醒 kswapd 以释放一些区的页。这个决定不是基于 `hysteresis` 的，而是当空闲页的数量低于 `watermark[WMARK_LOW]` 时就会进行；在这种情况下，`zone_wake_kswapd` 也被设置。

我所听到的（超棒的）想法：

1. 动态经历应该影响平衡：可以跟踪一个区的失败请求的数量，并反馈到平衡方案中 ([jalvo@mbay.net](mailto:jalvo@mbay.net))。
2. 实现一个类似于 `replace_with_highmem()` 的 `replace_with_regular()`，以保留 dma 页面。  
([lkd@tantalophile.demon.co.uk](mailto:lkd@tantalophile.demon.co.uk))

## DAMON: 数据访问监视器

DAMON 是 Linux 内核的一个数据访问监控框架系统。DAMON 的核心机制使其成为（该核心机制详见（设计））

- 准确度（监测输出对 DRAM 级别的内存管理足够有用；但可能不适合 CPU Cache 级别），
- 轻量级（监控开销低到可以在线应用），以及
- 可扩展（无论目标工作负载的大小，开销的上限值都在恒定范围内）。

因此，利用这个框架，内核的内存管理机制可以做出高级决策。会导致高数据访问监控开销的实验性内存管理优化工作可以再次进行。同时，在用户空间，有一些特殊工作负载的用户可以编写个性化应用程序，以便更好地了解和优化他们的工作负载和系统。

## 常见问题

### 为什么是一个新的子系统，而不是扩展 perf 或其他用户空间工具？

首先，因为它需要尽可能的轻量级，以便可以在线使用，所以应该避免任何不必要的开销，如内核-用户空间的上下文切换成本。第二，DAMON 的目标是被包括内核在内的其他程序所使用。因此，对特定工具（如 perf）的依赖性是不可取的。这就是 DAMON 在内核空间实现的两个最大的原因。

### “闲置页面跟踪”或“perf mem”可以替代 DAMON 吗？

闲置页跟踪是物理地址空间访问检查的一个低层次的原始方法。“perf mem”也是类似的，尽管它可以使用采样来减少开销。另一方面，DAMON 是一个更高层次的框架，用于监控各种地址空间。它专注于内存管理优化，并提供复杂的精度/开销处理机制。因此，“空闲页面跟踪”和“perf mem”可以提供 DAMON 输出的一个子集，但不能替代 DAMON。

### DAMON 是否只支持虚拟内存？

不，DAMON 的核心是独立于地址空间的。用户可以在 DAMON 核心上实现和配置特定地址空间的低级原始部分，包括监测目标区域的构造和实际的访问检查。通过这种方式，DAMON 用户可以用任何访问检查技术来监测任何地址空间。

尽管如此，DAMON 默认为虚拟内存和物理内存提供了基于 vma/rmap 跟踪和 PTE 访问位检查的地址空间相关功能的实现，以供参考和方便使用。

### 我可以简单地监测页面的粒度吗？

是的，你可以通过设置 `min_nr_regions` 属性高于工作集大小除以页面大小的值来实现。因为监视目标区域的大小被强制为 `>=page_size`，所以区域分割不会产生任何影响。

## 设计

### 可配置的层

DAMON 提供了数据访问监控功能，同时使其准确性和开销可控。基本的访问监控需要依赖于目标地址空间并为之优化的基元。另一方面，作为 DAMON 的核心，准确性和开销的权衡机制是在纯逻辑空间中。DAMON 将这两部分分离在不同的层中，并定义了它的接口，以允许各种低层次的基元实现与核心逻辑的配置。

由于这种分离的设计和可配置的接口，用户可以通过配置核心逻辑和适当的低级基元实现来扩展 DAMON 的任何地址空间。如果没有提供合适的，用户可以自己实现基元。

例如，物理内存、虚拟内存、交换空间、那些特定的进程、NUMA 节点、文件和支持的内存设备将被支持。另外，如果某些架构或设备支持特殊的优化访问检查基元，这些基元将很容易被配置。

## 特定地址空间基元的参考实现

基本访问监测的低级基元被定义为两部分。：

1. 确定地址空间的监测目标地址范围
2. 目标空间中特定地址范围的访问检查。

DAMON 目前为物理和虚拟地址空间提供了基元的实现。下面两个小节描述了这些工作的方式。

## 基于 VMA 的目标地址范围构造

这仅仅是针对虚拟地址空间基元的实现。对于物理地址空间，只是要求用户手动设置监控目标地址范围。

在进程的超级巨大的虚拟地址空间中，只有小部分被映射到物理内存并被访问。因此，跟踪未映射的地址区域只是一种浪费。然而，由于 DAMON 可以使用自适应区域调整机制来处理一定程度的噪声，所以严格来说，跟踪每一个映射并不是必须的，但在某些情况下甚至会产生很高的开销。也就是说，监测目标内部过于巨大的未映射区域应该被移除，以不占用自适应机制的时间。

出于这个原因，这个实现将复杂的映射转换为三个不同的区域，覆盖地址空间的每个映射区域。这三个区域之间的两个空隙是给定地址空间中两个最大的未映射区域。这两个最大的未映射区域是堆和最上面的 mmap() 区域之间的间隙，以及在大多数情况下最下面的 mmap() 区域和堆之间的间隙。因为这些间隙在通常的地址空间中是异常巨大的，排除这些间隙就足以做出合理的权衡。下面详细说明了这一点：

```
<heap>
<BIG UNMAPPED REGION 1>
<uppermost mmap()-ed region>
(small mmap()-ed regions and munmap()-ed regions)
<lowermost mmap()-ed region>
<BIG UNMAPPED REGION 2>
<stack>
```

## 基于 PTE 访问位的访问检查

物理和虚拟地址空间的实现都使用 PTE Accessed-bit 进行基本访问检查。唯一的区别在于从地址中找到相关的 PTE 访问位的方式。虚拟地址的实现是为该地址的目标任务查找页表，而物理地址的实现则是查找与该地址有映射关系的每一个页表。通过这种方式，实现者找到并清除下一个采样目标地址的位，并检查该位是否在一个采样周期后再次设置。这可能会干扰其他使用访问位的内核子系统，即空闲页跟踪和回收逻辑。为了避免这种干扰，DAMON 使其与空闲页面跟踪相互排斥，并使用 PG\_idle 和 PG\_young 页面标志来解决与回收逻辑的冲突，就像空闲页面跟踪那样。

## 独立于地址空间的核心机制

下面四个部分分别描述了 DAMON 的核心机制和五个监测属性，即 采样间隔、聚集间隔、更新间隔、最小区域数和 最大区域数。

### 访问频率监测

DAMON 的输出显示了在给定的时间内哪些页面的访问频率是多少。访问频率的分辨率是通过设置 采样间隔 和 聚集间隔 来控制的。详细地说，DAMON 检查每个 采样间隔 对每个页面的访问，并将结果汇总。换句话说，计算每个页面的访问次数。在每个 聚合间隔 过去后，DAMON 调用先前由用户注册的回调函数，以便用户可以阅读聚合的结果，然后再清除这些结果。这可以用以下简单的伪代码来描述：

```
while monitoring_on:
    for page in monitoring_target:
        if accessed(page):
            nr_accesses[page] += 1
    if time() % aggregation_interval == 0:
        for callback in user_registered_callbacks:
            callback(monitoring_target, nr_accesses)
        for page in monitoring_target:
            nr_accesses[page] = 0
    sleep(sampling_interval)
```

这种机制的监测开销将随着目标工作负载规模的增长而任意增加。

### 基于区域的抽样调查

为了避免开销的无限制增加，DAMON 将假定具有相同访问频率的相邻页面归入一个区域。只要保持这个假设（一个区域内的页面具有相同的访问频率），该区域内就只需要检查一个页面。因此，对于每个 采样间隔，DAMON 在每个区域中随机挑选一个页面，等待一个 采样间隔，检查该页面是否同时被访问，如果被访问则增加该区域的访问频率。因此，监测开销是可以通过设置区域的数量来控制的。DAMON 允许用户设置最小和最大的区域数量来进行权衡。

然而，如果假设没有得到保证，这个方案就不能保持输出的质量。

### 适应性区域调整

即使最初的监测目标区域被很好地构建以满足假设（同一区域内的页面具有相似的访问频率），数据访问模式也会被动态地改变。这将导致监测质量下降。为了尽可能地保持假设，DAMON 根据每个区域的访问频率自适应地进行合并和拆分。

对于每个 聚集区间，它比较相邻区域的访问频率，如果频率差异较小，就合并这些区域。然后，在它报告并清除每个区域的聚合接入频率后，如果区域总数不超过用户指定的最大区域数，它将每个区域拆分为两个或三个区域。

通过这种方式，DAMON 提供了其最佳的质量和最小的开销，同时保持了用户为其权衡设定的界限。

## 动态目标空间更新处理

监测目标地址范围可以动态改变。例如，虚拟内存可以动态地被映射和解映射。物理内存可以被热插拔。

由于在某些情况下变化可能相当频繁，DAMON 允许监控操作检查动态变化，包括内存映射变化，并仅在用户指定的时间间隔（更新间隔）中的每个时间段，将其应用于监控操作相关的数据结构，如抽象的监控目标内存存区。

## API 参考

内核空间的程序可以使用下面的 API 来使用 DAMON 的每个功能。你所需要做的就是引用 `damon.h`，它位于源代码树的 `include/linux/`。

### 结构体

该 API 在以下内核代码中：

`include/linux/damon.h`

### 函数

该 API 在以下内核代码中：

`mm/damon/core.c`

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:<alexs@kernel.org>)。

---

**Original** Documentation/vm/\_free\_page\_reporting.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译

### 空闲页报告

空闲页报告是一个 API，设备可以通过它来注册接收系统当前未使用的页面列表。这在虚拟化的情况下是很有用的，客户机能够使用这些数据来通知管理器它不再使用内存中的某些页面。

对于驱动，通常是气球驱动要使用这个功能，它将分配和初始化一个 `page_reporting_dev_info` 结构体。它要填充的结构体中的字段是用于处理散点列表的“report”函数指针。它还必须保证每次调用该函数时能处理至少相当于 `PAGE_REPORTING_CAPACITY` 的散点列表条目。假设没有其他页面报告设备已经注册，对 `page_reporting_register` 的调用将向报告框架注册页面报告接口。

一旦注册，页面报告 API 将开始向驱动报告成批的页面。API 将在接口被注册后 2 秒开始报告页面，并在任何足够高的页面被释放之后 2 秒继续报告。

报告的页面将被存储在传递给报告函数的散列表中，最后一个条目的结束位被设置在条目 `next-1` 中。当页面被报告函数处理时，分配器将无法访问它们。一旦报告函数完成，这些页将被返回到它们所获得的自由区域。

在移除使用空闲页报告的驱动之前，有必要调用 `page_reporting_unregister`，以移除目前被空闲页报告使用的 `page_reporting_dev_info` 结构体。这样做将阻止进一步的报告通过该接口发出。如果另一个驱动或同一驱动被注册，它就有可能恢复前一个驱动在报告空闲页方面的工作。

Alexander Duyck, 2019 年 12 月 04 日

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:<alexs@kernel.org>)。

---

**Original** Documentation/vm/highmem.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译

### 高内存处理

作者: Peter Zijlstra <[a.p.zijlstra@chello.nl](mailto:a.p.zijlstra@chello.nl)>

- 高内存是什么？
- 临时虚拟映射
- 使用 `kmap_atomic`
- 临时映射的成本

- *i386 PAE*

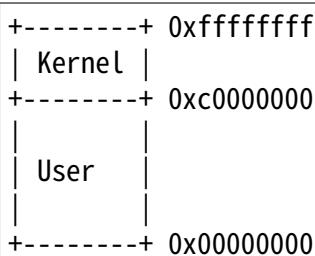
## 高内存是什么？

当物理内存的大小接近或超过虚拟内存的最大大小时，就会使用高内存（highmem）。在这一点上，内核不可能在任何时候都保持所有可用的物理内存的映射。这意味着内核需要开始使用它想访问的物理内存的临时映射。

没有被永久映射覆盖的那部分（物理）内存就是我们所说的“高内存”。对于这个边界的确切位置，有各种架构上的限制。

例如，在 i386 架构中，我们选择将内核映射到每个进程的虚拟空间，这样我们就不必为内核的进入/退出付出全部的 TLB 作废代价。这意味着可用的虚拟内存空间（i386 上为 4GiB）必须在用户和内核空间之间进行划分。

使用这种方法的架构的传统分配方式是 3:1，3GiB 用于用户空间，顶部的 1GiB 用于内核空间。：



这意味着内核在任何时候最多可以映射 1GiB 的物理内存，但是由于我们需要虚拟地址空间来做其他事情-包括访问其余物理内存的临时映射-实际的直接映射通常会更少（通常在 ~896MiB 左右）。

其他有 mm 上下文标签的 TLB 的架构可以有独立的内核和用户映射。然而，一些硬件（如一些 ARM）在使用 mm 上下文标签时，其虚拟空间有限。

## 临时虚拟映射

内核包含几种创建临时映射的方法。：

- `vmap()`. 这可以用来将多个物理页长期映射到一个连续的虚拟空间。它需要 `synchronization` 来解除映射。
- `kmap()`. 这允许对单个页面进行短期映射。它需要 `synchronization`，但在一定程度上被摊销。当以嵌套方式使用时，它也很容易出现死锁，因此不建议在新代码中使用它。
- `kmap_atomic()`. 这允许对单个页面进行非常短的时间映射。由于映射被限制在发布它的 CPU 上，它表现得很好，但发布任务因此被要求留在该 CPU 上直到它完成，以免其他任务取代它的映射。

`kmap_atomic()` 也可以由中断上下文使用，因为它不睡眠，而且调用者可能在调用 `kunmap_atomic()` 之后才睡眠。

可以假设 `k[un]map_atomic()` 不会失败。

## 使用 kmap\_atomic

何时何地使用 `kmap_atomic()` 是很直接的。当代码想要访问一个可能从高内存（见 `_GFP_HIGHMEM`）分配的页面的内容时，例如在页缓存中的页面，就会使用它。该 API 有两个函数，它们的使用方式与下面类似：

```
/* 找到感兴趣的页面。 */
struct page *page = find_get_page(mapping, offset);

/* 获得对该页内容的访问权。 */
void *vaddr = kmap_atomic(page);

/* 对该页的内容做一些处理。 */
memset(vaddr, 0, PAGE_SIZE);

/* 解除该页面的映射。 */
kunmap_atomic(vaddr);
```

注意，`kunmap_atomic()` 调用的是 `kmap_atomic()` 调用的结果而不是参数。

如果你需要映射两个页面，因为你想从一个页面复制到另一个页面，你需要保持 `kmap_atomic` 调用严格嵌套，如：

```
vaddr1 = kmap_atomic(page1);
vaddr2 = kmap_atomic(page2);

memcpy(vaddr1, vaddr2, PAGE_SIZE);

kunmap_atomic(vaddr2);
kunmap_atomic(vaddr1);
```

## 临时映射的成本

创建临时映射的代价可能相当高。体系架构必须操作内核的页表、数据 TLB 和/或 MMU 的寄存器。

如果 `CONFIG_HIGHMEM` 没有被设置，那么内核会尝试用一点计算来创建映射，将页面结构地址转换成指向页面内容的指针，而不是去捣鼓映射。在这种情况下，解映射操作可能是一个空操作。

如果 `CONFIG_MMU` 没有被设置，那么就不可能有临时映射和高内存。在这种情况下，也将使用计算方法。

## i386 PAE

在某些情况下，i386 架构将允许你在 32 位机器上安装多达 64GiB 的内存。但这有一些后果：

- Linux 需要为系统中的每个页面建立一个页帧结构，而且页帧需要驻在永久映射中，这意味着：
- 你最多可以有  $896M/\text{sizeof(struct page)}$  页帧；由于页结构体是 32 字节的，所以最终会有 112G 的页；然而，内核需要在内存中存储更多的页帧……
- PAE 使你的页表变大-这使系统变慢，因为更多的数据需要在 TLB 填充等方面被访问。一个好处是，PAE 有更多的 PTE 位，可以提供像 NX 和 PAT 这样的高级功能。

一般的建议是，你不要在 32 位机器上使用超过 8GiB 的空间-尽管更多的空间可能对你和你的工作量有用，但你几乎是靠你自己-不要指望内核开发者真的会很关心事情的进展情况。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

**Original** Documentation/vm/ksm.rst

翻译 徐鑫 xu xin <[xu.xin16@zte.com.cn](mailto:xu.xin16@zte.com.cn)>

## 内核同页合并

KSM 是一种节省内存的数据去重功能，由 CONFIG\_KSM=y 启用，并在 2.6.32 版本时被添加到 Linux 内核。详见 mm/ksm.c 的实现，以及 <http://lwn.net/Articles/306704> 和 <https://lwn.net/Articles/330589>

KSM 的用户空间的接口在 Documentation/translations/zh\_CN/admin-guide/mm/ksm.rst 文档中有描述。

## 设计

## 概述

概述内容请见 mm/ksm.c 文档中的“DOC: Overview”

## 逆映射

KSM 维护着稳定树中的 KSM 页的逆映射信息。

当 KSM 页面的共享数小于 `max_page_sharing` 的虚拟内存区域 (VMAs) 时，则代表了 KSM 页的稳定树其中的节点指向了一个 `rmap_item` 结构体类型的列表。同时，这个 KSM 页的 `page->mapping` 指向了该稳定树节点。

如果共享数超过了阈值，KSM 将给稳定树添加第二个维度。稳定树就变成链接一个或多个稳定树”副本”的”链”。每个副本都保留 KSM 页的逆映射信息，其中 `page->mapping` 指向该”副本”。

每个链以及链接到该链中的所有”副本”强制不变的是，它们代表了相同的写保护内存内容，尽管任中一个”副本”是由同一片内存区的不同的 KSM 复制页所指向的。

这样一来，相比与无限的逆映射链表，稳定树的查找计算复杂性不受影响。但在稳定树本身中不能有重复的 KSM 页面内容仍然是强制要求。

由 `max_page_sharing` 强制决定的数据去重限制是必要的，以此来避免虚拟内存 `rmap` 链表变得过大。`rmap` 的遍历具有  $O(N)$  的复杂度，其中  $N$  是共享页面的 `rmap` 项（即虚拟映射）的数量，而这个共享页面的节点数量又被 `max_page_sharing` 所限制。因此，这有效地将线性  $O(N)$  计算复杂度从 `rmap` 遍历中分散到不同的 KSM 页面上。`ksmd` 进程在稳定节点”链”上的遍历也是  $O(N)$ ，但这个  $N$  是稳定树”副本”的数量，而不是 `rmap` 项的数量，因此它对 `ksmd` 性能没有显著影响。实际上，最佳稳定树”副本”的候选节点将保留在”副本”列表的开头。

`max_page_sharing` 的值设置得高了会促使更快的内存合并（因为将有更少的稳定树副本排队进入稳定节点 `chain->hlist`）和更高的数据去重系数，但代价是在交换、压缩、NUMA 平衡和页面迁移过程中可能导致 KSM 页的最大 `rmap` 遍历速度较慢。

`stable_node_dups/stable_node_chains` 的比值还受 `max_page_sharing` 调控的影响，高比值可能意味着稳定节点 `dup` 中存在碎片，这可以通过在 `ksmd` 中引入碎片算法来解决，该算法将 `rmap` 项从一个稳定节点 `dup` 重定位到另一个稳定节点 `dup`，以便释放那些仅包含极少 `rmap` 项的稳定节点”`dup`”，但这可能会增加 `ksmd` 进程的 CPU 使用率，并可能会减慢应用程序在 KSM 页面上的只读计算。

KSM 会定期扫描稳定节点”链”中链接的所有稳定树”副本”，以便删减过时了的稳定节点。这种扫描的频率由 `stable_node_chains_prune_millisecs` 这个 sysfs 接口定义。

## 参考

内核代码请见 mm/ksm.c。涉及的函数 (mm\_slot ksm\_scan stable\_node rmap\_item)。

## Frontswap

Frontswap 为交换页提供了一个“transcendent memory”的接口。在一些环境中，由于交换页被保存在 RAM（或类似 RAM 的设备）中，而不是交换磁盘，因此可以获得巨大的性能节省（提高）。

Frontswap 之所以这么命名，是因为它可以被认为是与 swap 设备的“back”存储相反。存储器被认为是一个同步并发安全的面向页面的“伪 RAM 设备”，符合 transcendent memory（如 Xen 的“tmem”，或内核内压缩内存，又称“zcache”，或未来的类似 RAM 的设备）的要求；这个伪 RAM 设备不能被内核直接访问或寻址，其大小未知且可能随时间变化。驱动程序通过调用 frontswap\_register\_ops 将自己与 frontswap 链接起来，以适当地设置 frontswap\_ops 的功能，它提供的功能必须符合某些策略，如下所示：

一个“init”将设备准备好接收与指定的交换设备编号（又称“类型”）相关的 frontswap 交换页。一个“store”将把该页复制到 transcendent memory，并与该页的类型和偏移量相关联。一个“load”将把该页，如果找到的话，从 transcendent memory 复制到内核内存，但不会从 transcendent memory 中删除该页。一个“invalidate\_page”将从 transcendent memory 中删除该页，一个“invalidate\_area”将删除所有与交换类型相关的页（例如，像 swapoff）并通知“device”拒绝进一步存储该交换类型。

一旦一个页面被成功存储，在该页面上的匹配加载通常会成功。因此，当内核发现自己处于需要交换页面的情况时，它首先尝试使用 frontswap。如果存储的结果是成功的，那么数据就已经成功的保存到了 transcendent memory 中，并且避免了磁盘写入，如果后来再读回数据，也避免了磁盘读取。如果存储返回失败，transcendent memory 已经拒绝了该数据，且该页可以像往常一样被写入交换空间。

请注意，如果一个页面被存储，而该页面已经存在于 transcendent memory 中（一个“重复”的存储），要么存储成功，数据被覆盖，要么存储失败，该页面被废止。这确保了旧的数据永远不会从 frontswap 中获得。如果配置正确，对 frontswap 的监控是通过 /sys/kernel/debug/frontswap 目录下的 debugfs 完成的。frontswap 的有效性可以通过以下方式测量（在所有交换设备中）：

**failed\_stores** 有多少次存储的尝试是失败的

**loads** 尝试了多少次加载（应该全部成功）

**succ\_stores** 有多少次存储的尝试是成功的

**invalidates** 尝试了多少次作废

后台实现可以提供额外的指标。

## 经常问到的问题

- 价值在哪里？

当一个工作负载开始交换时，性能就会下降。Frontswap 通过提供一个干净的、动态的接口来读取和写入交换页到“transcendent memory”，从而大大增加了许多这样的工作负载的性能，否则内核是无法直接寻址的。当数据被转换为不同的形式和大小（比如压缩）或者被秘密移动（对于一些类似 RAM 的设备来说，这可能对写平衡很有用）时，这个接口是理想的。交换页（和被驱逐的页面缓存页）是这种比 RAM 慢但比磁盘快得多的“伪 RAM 设备”的一大用途。

Frontswap 对内核的影响相当小，为各种系统配置中更动态、更灵活的 RAM 利用提供了巨大的灵活性：

在单一内核的情况下，又称“zcache”，页面被压缩并存储在本地内存中，从而增加了可以安全保存在 RAM 中的匿名页面总数。Zcache 本质上是用压缩/解压缩的 CPU 周期换取更好的内存利用率。Benchmarks 测试显示，当内存压力较低时，几乎没有影响，而在高内存压力下的一些工作负载上，则有明显的性能改善（25% 以上）。

“RAMster”在 zcache 的基础上增加了对集群系统的“peer-to-peer”transcendent memory 的支持。Frontswap 页面像 zcache 一样被本地压缩，但随后被“remotified”到另一个系统的 RAM。这使得 RAM 可以根据需要动态地来回负载平衡，也就是说，当系统 A 超载时，它可以交换到系统 B，反之亦然。RAMster 也可以被配置成一个内存服务器，因此集群中的许多服务器可以根据需要动态地交换到配置有大量内存的单一服务器上……而不需要预先配置每个客户有多少内存可用

在虚拟情况下，虚拟化的全部意义在于统计地将物理资源在多个虚拟机的不同需求之间进行复用。对于 RAM 来说，这真的很难做到，而且在不改变内核的情况下，要做好这一点的努力基本上是失败的（除了一些广为人知的特殊情况下的工作负载）。具体来说，Xen Transcendent Memory 后端允许管理器拥有的 RAM “fallow”，不仅可以在多个虚拟机之间进行“time-shared”，而且页面可以被压缩和重复利用，以优化 RAM 的利用率。当客户操作系统被诱导交出未充分利用的 RAM 时（如“selfballooning”），突然出现的意外内存压力可能会导致交换；frontswap 允许这些页面被交换到管理器 RAM 中或从管理器 RAM 中交换（如果整体主机系统内存条件允许），从而减轻计划外交换可能带来的可怕的性能影响。

一个 KVM 的实现正在进行中，并且已经被 RFC’ed 到 lkml。而且，利用 frontswap，对 NVM 作为内存扩展技术的调查也在进行中。

- 当然，在某些情况下可能有性能上的优势，但 frontswap 的空间/时间开销是多少？

如果 CONFIG\_FRONTSWAP 被禁用，每个 frontswap 钩子都会编译成空，唯一的开销是每个 swapon’ed swap 设备的几个额外字节。如果 CONFIG\_FRONTSWAP 被启用，但没有 frontswap 的“backend”寄存器，每读或写一个交换页就会有一个额外的全局变量，而不是零。如果 CONFIG\_FRONTSWAP 被启用，并且有一个 frontswap 的 backend 寄存器，并且后端每次“store”请求都失败（即尽管声称可能，但没有提供内存），CPU 的开销仍然可以忽略不计 - 因为每次 frontswap 失败都是在交换页写到磁盘之前，系统很可能是 I/O 绑定的，无论如何使用一小部分的 CPU 都是不相关的。

至于空间，如果 CONFIG\_FRONTSWAP 被启用，并且有一个 frontswap 的 backend 注册，那么每个交换设备的每个交换页都会被分配一个比特。这是在内核已经为每个交换设备的每个交换页分配的 8 位（在 2.6.34 之前是 16 位）上增加的。（Hugh Dickins 观察到，frontswap 可能会偷取现有的 8 个比特，但是我们以后再来担心这个小的优化问题）。对于标准的 4K 页面大小的非常大的交换盘（这很罕见），这是每 32GB

交换盘 1MB 开销。

当交换页存储在 `transcendent memory` 中而不是写到磁盘上时，有一个副作用，即这可能会产生更多的内存压力，有可能超过其他的优点。一个 `backend`，比如 `zcache`，必须实现策略来仔细（但动态地）管理内存限制，以确保这种情况不会发生。

- 好吧，那就用内核骇客能理解的术语来快速概述一下这个 `frontswap` 补丁的作用如何？

我们假设在内核初始化过程中，一个 `frontswap` 的“`backend`”已经注册了；这个注册表明这个 `frontswap` 的“`backend`”可以访问一些不被内核直接访问的“内存”。它到底提供了多少内存是完全动态和随机的。

每当一个交换设备被交换时，就会调用 `frontswap_init()`，把交换设备的编号（又称“类型”）作为一个参数传给它。这就通知了 `frontswap`，以期待“`store`”与该号码相关的交换页的尝试。

每当交换子系统准备将一个页面写入交换设备时（参见 `swap_writepage()`），就会调用 `frontswap_store`。`Frontswap` 与 `frontswap backend` 协商，如果 `backend` 说它没有空间，`frontswap_store` 返回-1，内核就会照常把页换到交换设备上。注意，来自 `frontswap backend` 的响应对内核来说是不可预测的；它可能选择从不接受一个页面，可能接受每九个页面，也可能接受每一个页面。但是如果 `backend` 确实接受了一个页面，那么这个页面的数据已经被复制并与类型和偏移量相关联了，而且 `backend` 保证了数据的持久性。在这种情况下，`frontswap` 在交换设备的“`frontswap_map`”中设置了一个位，对应于交换设备上的页面偏移量，否则它就会将数据写入该设备。

当交换子系统需要交换一个页面时（`swap_readpage()`），它首先调用 `frontswap_load()`，检查 `frontswap_map`，看这个页面是否早先被 `frontswap backend` 接受。如果是，该页的数据就会从 `frontswap` 后端填充，换入就完成了。如果不是，正常的交换代码将被执行，以便从真正的交换设备上获得这一页的数据。

所以每次 `frontswap backend` 接受一个页面时，交换设备的读取和（可能）交换设备的写入都被“`frontswap backend store`”和（可能）“`frontswap backend loads`”所取代，这可能会快得多。

- `frontswap` 不能被配置为一个“特殊的”交换设备，它的优先级要高于任何真正的交换设备（例如像 `zswap`，或者可能是 `swap-over-nbd/NFS`）？

首先，现有的交换子系统不允许有任何种类的交换层次结构。也许它可以被重写以适应层次结构，但这将需要相当大的改变。即使它被重写，现有的交换子系统也使用了块 I/O 层，它假定交换设备是固定大小的，其中的任何页面都是可线性寻址的。`Frontswap` 几乎没有触及现有的交换子系统，而是围绕着块 I/O 子系统的限制，提供了大量的灵活性和动态性。

例如，`frontswap backend` 对任何交换页的接受是完全不可预测的。这对 `frontswap backend` 的定义至关重要，因为它赋予了 `backend` 完全动态的决定权。在 `zcache` 中，人们无法预先知道一个页面的可压缩性如何。可压缩性“差”的页面会被拒绝，而“差”本身也可以根据当前的内存限制动态地定义。

此外，`frontswap` 是完全同步的，而真正的交换设备，根据定义，是异步的，并且使用块 I/O。块 I/O 层不仅是不必要的，而且可能进行“优化”，这对面向 RAM 的设备来说是不合适的，包括将一些页面的写入延迟相当长的时间。同步是必须的，以确保后端的动态性，并避免棘手的竞争条件，这将不必要地大大增加 `frontswap` 和/或块 I/O 子系统的复杂性。也就是说，只有最初的“`store`”和“`load`”操作是需要同步的。一个独立的异步线程可以自由地操作由 `frontswap` 存储的页面。例如，`RAMster` 中的“`remotification`”线程使用标准的异步内核套接字，将压缩的 `frontswap` 页面移动到远程机器。同样，KVM 的客户方实现可以进行客户内压缩，并使用“batched” hypercalls。

在虚拟化环境中，动态性允许管理程序（或主机操作系统）做“intelligent overcommit”。例如，它可以选择只接受页面，直到主机交换可能即将发生，然后强迫客户机做他们自己的交换。

transcendent memory 规格的 frontswap 有一个坏处。因为任何“store”都可能失败，所以必须在一个真正的交换设备上有一个真正的插槽来交换页面。因此，frontswap 必须作为每个交换设备的“影子”来实现，它有可能容纳交换设备可能容纳的每一个页面，也有可能根本不容纳任何页面。这意味着 frontswap 不能包含比 swap 设备总数更多的页面。例如，如果在某些安装上没有配置交换设备，frontswap 就没有用。无交换设备的便携式设备仍然可以使用 frontswap，但是这种设备的 backend 必须配置某种“ghost”交换设备，并确保它永远不会被使用。

- 为什么会有这种关于“重复存储”的奇怪定义？如果一个页面以前被成功地存储过，难道它不能总是被成功地覆盖吗？

几乎总是可以的，不，有时不能。考虑一个例子，数据被压缩了，原来的 4K 页面被压缩到了 1K。现在，有人试图用不可压缩的数据覆盖该页，因此会占用整个 4K。但是 backend 没有更多的空间了。在这种情况下，这个存储必须被拒绝。每当 frontswap 拒绝一个会覆盖的存储时，它也必须使旧的数据作废，并确保它不再被访问。因为交换子系统会把新的数据写到读交换设备上，这是确保一致性的正确做法。

- 为什么 frontswap 补丁会创建新的头文件 swapfile.h？

frontswap 代码依赖于一些 swap 子系统内部的数据结构，这些数据结构多年来一直在静态和全局之间来回移动。这似乎是一个合理的妥协：将它们定义为全局，但在一个新的包含文件中声明它们，该文件不被包含 swap.h 的大量源文件所包含。

Dan Magenheimer，最后更新于 2012 年 4 月 9 日

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:<alexs@kernel.org>)。

---

**Original** Documentation/vm/hmm.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译

## 异构内存管理 (HMM)

提供基础设施和帮助程序以将非常规内存（设备内存，如板上 GPU 内存）集成到常规内核路径中，其基石是此类内存的专用 struct page（请参阅本文档的第 5 至 7 节）。

HMM 还为 SVM（共享虚拟内存）提供了可选的帮助程序，即允许设备透明地访问与 CPU 一致的程序地址，这意味着 CPU 上的任何有效指针也是该设备的有效指针。这对于简化高级异构计算的使用变得必不可少，其中 GPU、DSP 或 FPGA 用于代表进程执行各种计算。

本文档分为以下部分：在第一部分中，我揭示了与使用特定于设备的内存分配器相关的问题。在第二部分中，我揭示了许多平台固有的硬件限制。第三部分概述了 HMM 设计。第四部分解释了 CPU 页表镜像的工作原理以及 HMM 在这种情况下的目的。第五部分处理内核中如何表示设备内存。最后，最后一节介绍了一个新的迁移助手，它允许利用设备 DMA 引擎。

- 使用特定于设备的内存分配器的问题
- I/O 总线、设备内存特性
- 共享地址空间和迁移
- 地址空间镜像实现和 API
- 利用 `default_flags` 和 `pfn_flags_mask`
- 从核心内核的角度表示和管理设备内存
- 移入和移出设备内存
- 独占访问存储器
- 内存 `cgroup (memcg)` 和 rss 统计

## 使用特定于设备的内存分配器的问题

具有大量板载内存（几 GB）的设备（如 GPU）历来通过专用驱动程序特定 API 管理其内存。这会造成设备驱动程序分配和管理的内存与常规应用程序内存（私有匿名、共享内存或常规文件支持内存）之间的隔断。从这里开始，我将把这个方面称为分割的地址空间。我使用共享地址空间来指代相反的情况：即，设备可以透明地使用任何应用程序内存区域。

分割的地址空间的发生是因为设备只能访问通过设备特定 API 分配的内存。这意味着从设备的角度来看，程序中的所有内存对象并不平等，这使得依赖于广泛的库的大型程序变得复杂。

具体来说，这意味着想要利用像 GPU 这样的设备的代码需要在通用分配的内存（`malloc`、`mmap` 私有、`mmap` 共享）和通过设备驱动程序 API 分配的内存之间复制对象（这仍然以 `mmap` 结束，但是是设备文件）。

对于平面数据集（数组、网格、图像……），这并不难实现，但对于复杂数据集（列表、树……），很难做到正确。复制一个复杂的数据集需要重新映射其每个元素之间的所有指针关系。这很容易出错，而且由于数据集和地址的重复，程序更难调试。

分割地址空间也意味着库不能透明地使用它们从核心程序或另一个库中获得的数据，因此每个库可能不得不使用设备特定的内存分配器来重复其输入数据集。大型项目会因此受到影响，并因为各种内存拷贝而浪费资源。

复制每个库的 API 以接受每个设备特定分配器分配的内存作为输入或输出，并不是一个可行的选择。这将导致库入口点的组合爆炸。

最后，随着高级语言结构（在 C++ 中，当然也在其他语言中）的进步，编译器现在有可能在没有程序员干预的情况下利用 GPU 和其他设备。某些编译器识别的模式仅适用于共享地址空间。对所有其他模式，使用共享地址空间也更合理。

### I/O 总线、设备内存特性

由于一些限制，I/O 总线削弱了共享地址空间。大多数 I/O 总线只允许从设备到主内存的基本内存访问；甚至缓存一致性通常是可选的。从 CPU 访问设备内存甚至更加有限。通常情况下，它不是缓存一致的。

如果我们只考虑 PCIE 总线，那么设备可以访问主内存（通常通过 IOMMU）并与 CPU 缓存一致。但是，它只允许设备对主存储器进行一组有限的原子操作。这在另一个方向上更糟：CPU 只能访问有限范围的设备内存，而不能对其执行原子操作。因此，从内核的角度来看，设备内存不能被视为与常规内存等同。

另一个严重的因素是带宽有限（约 32GBytes/s，PCIE 4.0 和 16 通道）。这比最快的 GPU 内存（1 TBytes/s）慢 33 倍。最后一个限制是延迟。从设备访问主内存的延迟比设备访问自己的内存时高一个数量级。

一些平台正在开发新的 I/O 总线或对 PCIE 的添加/修改以解决其中一些限制（OpenCAPI、CCIX）。它们主要允许 CPU 和设备之间的双向缓存一致性，并允许架构支持的所有原子操作。遗憾的是，并非所有平台都遵循这一趋势，并且一些主要架构没有针对这些问题的硬件解决方案。

因此，为了使共享地址空间有意义，我们不仅必须允许设备访问任何内存，而且还必须允许任何内存设备使用时迁移到设备内存（在迁移时阻止 CPU 访问）。

### 共享地址空间和迁移

HMM 打算提供两个主要功能。第一个是通过复制 cpu 页表到设备页表中来共享地址空间，因此对于进程地址空间中的任何有效主内存地址，相同的地址指向相同的物理内存。

为了实现这一点，HMM 提供了一组帮助程序来填充设备页表，同时跟踪 CPU 页表更新。设备页表更新不像 CPU 页表更新那么容易。要更新设备页表，您必须分配一个缓冲区（或使用预先分配的缓冲区池）并在其中写入 GPU 特定命令以执行更新（取消映射、缓存失效和刷新等）。这不能通过所有设备的通用代码来完成。因此，为什么 HMM 提供了帮助器，在把硬件的具体细节留给设备驱动程序的同时，把一切可以考虑的因素都考虑进去了。

HMM 提供的第二种机制是一种新的 ZONE\_DEVICE 内存，它允许为设备内存的每个页面分配一个 struct page。这些页面很特殊，因为 CPU 无法映射它们。然而，它们允许使用现有的迁移机制将主内存迁移到设备内存，从 CPU 的角度来看，一切看起来都像是换出到磁盘的页面。使用 struct page 可以与现有的 mm 机制进行最简单、最干净的集成。再次，HMM 仅提供帮助程序，首先为设备内存热插拔新的 ZONE\_DEVICE 内存，然后执行迁移。迁移内容和时间的策略决定留给设备驱动程序。

请注意，任何 CPU 对设备页面的访问都会触发缺页异常并迁移回主内存。例如，当支持给定 CPU 地址 A 的页面从主内存页面迁移到设备页面时，对地址 A 的任何 CPU 访问都会触发缺页异常并启动向主内存的迁移。

凭借这两个特性，HMM 不仅允许设备镜像进程地址空间并保持 CPU 和设备页表同步，而且还通过迁移设备正在使用的数据集部分来利用设备内存。

## 地址空间镜像实现和 API

地址空间镜像的主要目标是允许将一定范围的 CPU 页表复制到一个设备页表中；HMM 有助于保持两者同步。想要镜像进程地址空间的设备驱动程序必须从注册 mmu\_interval\_notifier 开始：

```
int mmu_interval_notifier_insert(struct mmu_interval_notifier *interval_sub,
                                 struct mm_struct *mm, unsigned long start,
                                 unsigned long length,
                                 const struct mmu_interval_notifier_ops *ops);
```

在 ops->invalidate() 回调期间，设备驱动程序必须对范围执行更新操作（将范围标记为只读，或完全取消映射等）。设备必须在驱动程序回调返回之前完成更新。

当设备驱动程序想要填充一个虚拟地址范围时，它可以使用：

```
int hmm_range_fault(struct hmm_range *range);
```

如果请求写访问，它将在丢失或只读条目上触发缺页异常（见下文）。缺页异常使用通用的 mm 缺页异常代码路径，就像 CPU 缺页异常一样。

这两个函数都将 CPU 页表条目复制到它们的 pfns 数组参数中。该数组中的每个条目对应于虚拟范围中的一个地址。HMM 提供了一组标志来帮助驱动程序识别特殊的 CPU 页表项。

在 sync\_cpu\_device\_pagetables() 回调中锁定是驱动程序必须尊重的最重要的方面，以保持事物正确同步。使用模式是：

```
int driver_populate_range(...)
{
    struct hmm_range range;
    ...

    range.notifier = &interval_sub;
    range.start = ...;
    range.end = ...;
    range.hmm_pfns = ...;

    if (!mmget_not_zero(interval_sub->notifier.mm))
        return -EFAULT;

again:
    range.notifier_seq = mmu_interval_read_begin(&interval_sub);
    mmap_read_lock(mm);
    ret = hmm_range_fault(&range);
}
```

```

if (ret) {
    mmap_read_unlock(mm);
    if (ret == -EBUSY)
        goto again;
    return ret;
}
mmap_read_unlock(mm);

take_lock(driver->update);
if (mmu_interval_read_retry(&ni, range.notifier_seq) {
    release_lock(driver->update);
    goto again;
}

/* Use pfns array content to update device page table,
 * under the update lock */
release_lock(driver->update);
return 0;
}

```

driver->update 锁与驱动程序在其 invalidate() 回调中使用的锁相同。该锁必须在调用 mmu\_interval\_read\_retry() 之前保持，以避免与并发 CPU 页表更新发生任何竞争。

## 利用 default\_flags 和 pfn\_flags\_mask

hmm\_range 结构有 2 个字段，default\_flags 和 pfn\_flags\_mask，它们指定整个范围的故障或快照策略，而不必为 pfns 数组中的每个条目设置它们。

例如，如果设备驱动程序需要至少具有读取权限的范围的页面，它会设置：

```

range->default_flags = HMM_PFN_REQ_FAULT;
range->pfn_flags_mask = 0;

```

并如上所述调用 hmm\_range\_fault()。这将填充至少具有读取权限的范围内的所有页面。

现在假设驱动程序想要做同样的事情，除了它想要拥有写权限的范围内一页。现在驱动程序设置：

```

range->default_flags = HMM_PFN_REQ_FAULT;
range->pfn_flags_mask = HMM_PFN_REQ_WRITE;
range->pfns[index_of_write] = HMM_PFN_REQ_WRITE;

```

有了这个，HMM 将在至少读取（即有效）的所有页面中异常，并且对于地址 == range->start + (index\_of\_write << PAGE\_SHIFT) 它将异常写入权限，即，如果 CPU pte 没有设置写权限，那么 HMM 将调用 handle\_mm\_fault()。

hmm\_range\_fault 完成后，标志位被设置为页表的当前状态，即 HMM\_PFN\_VALID | 如果页面可写，将设置 HMM\_PFN\_WRITE。

## 从核心内核的角度表示和管理设备内存

尝试了几种不同的设计来支持设备内存。第一个使用特定于设备的数据结构来保存有关迁移内存的信息，HMM 将自身挂接到 mm 代码的各个位置，以处理对设备内存支持的地址的任何访问。事实证明，这最终复制了 struct page 的大部分字段，并且还需要更新许多内核代码路径才能理解这种新的内存类型。

大多数内核代码路径从不尝试访问页面后面的内存，而只关心 struct page 的内容。正因为如此，HMM 切换到直接使用 struct page 用于设备内存，这使得大多数内核代码路径不知道差异。我们只需要确保没有人试图从 CPU 端映射这些页面。

## 移入和移出设备内存

由于 CPU 无法直接访问设备内存，因此设备驱动程序必须使用硬件 DMA 或设备特定的加载/存储指令来迁移数据。migrate\_vma\_setup()、migrate\_vma\_pages() 和 migrate\_vma\_finalize() 函数旨在使驱动程序更易于编写并集中跨驱动程序的通用代码。

在将页面迁移到设备私有内存之前，需要创建特殊的设备私有 struct page。这些将用作特殊的“交换”页表条目，以便 CPU 进程在尝试访问已迁移到设备专用内存的页面时会发生异常。

这些可以通过以下方式分配和释放：

```
struct resource *res;
struct dev_pagemap pagemap;

res = request_free_mem_region(&iomem_resource, /* number of bytes */,
                             "name of driver resource");
pagemap.type = MEMORY_DEVICE_PRIVATE;
pagemap.range.start = res->start;
pagemap.range.end = res->end;
pagemap.nr_range = 1;
pagemap.ops = &device_devmem_ops;
memremap_pages(&pagemap, numa_node_id());

memunmap_pages(&pagemap);
release_mem_region(pagemap.range.start, range_len(&pagemap.range));
```

还有 devm\_request\_free\_mem\_region()、devm\_memremap\_pages()、devm\_memunmap\_pages() 和 devm\_release\_mem\_region() 当资源可以绑定到 struct device.

整体迁移步骤类似于在系统内存中迁移 NUMA 页面 (see Page migration)，但这些步骤分为设备驱动程序特定代码和共享公共代码：

1. mmap\_read\_lock()

设备驱动程序必须将 struct vm\_area\_struct 传递给 migrate\_vma\_setup()，因此需要在迁移期间保留 mmap\_read\_lock() 或 mmap\_write\_lock()。

2. migrate\_vma\_setup(struct migrate\_vma \*args)

设备驱动初始化了 `struct migrate_vma` 的字段，并将该指针传递给 `migrate_vma_setup()`。`args->flags` 字段是用来过滤哪些源页面应该被迁移。例如，设置 `MIGRATE_VMA_SELECT_SYSTEM` 将只迁移系统内存，设置 `MIGRATE_VMA_SELECT_DEVICE_PRIVATE` 将只迁移驻留在设备私有内存中的页面。如果后者被设置，`args->pgmap_owner` 字段被用来识别驱动所拥有的设备私有页。这就避免了试图迁移驻留在其他设备中的设备私有页。目前，只有匿名的私有 VMA 范围可以被迁移到系统内存和设备私有内存。

`migrate_vma_setup()` 所做的第一步是用 `mmu_notifier_invalidate_range_start()` 和 `mmu_notifier_invalidate_range_end()` 调用来遍历设备周围的页表，使其他设备的 MMU 无效，以便在 `args->src` 数组中填写要迁移的 PFN。`invalidate_range_start()` 回调传递给一个```struct mmu_notifier_range```，其 `event` 字段设置为 `MMU_NOTIFY_MIGRATE`，`owner` 字段设置为传递给 `migrate_vma_setup()` 的 `args->pgmap_owner` 字段。这允许设备驱动跳过无效化回调，只无效化那些实际正在迁移的设备私有 MMU 映射。这一点将在下一节详细解释。

在遍历页表时，一个 `pte_none()` 或 `is_zero_pfn()` 条目导致一个有效的“zero”PFN 存储在 `args->src` 阵列中。这让驱动分配设备私有内存并清除它，而不是复制一个零页。到系统内存或设备私有结构页的有效 PTE 条目将被 `lock_page()` 锁定，与 LRU 隔离（如果系统内存和设备私有页不在 LRU 上），从进程中取消映射，并插入一个特殊的迁移 PTE 来代替原来的 PTE。`migrate_vma_setup()` 还清除了 ```args->dst` 数组。

### 3. 设备驱动程序分配目标页面并将源页面复制到目标页面。

驱动程序检查每个 `src` 条目以查看该 `MIGRATE_PFN_MIGRATE` 位是否已设置并跳过未迁移的条目。设备驱动程序还可以通过不填充页面的 `dst` 数组来选择跳过页面迁移。

然后，驱动程序分配一个设备私有 `struct page` 或一个系统内存页，用 `lock_page()` 锁定该页，并将 `dst` 数组条目填入：

```
dst[i] = migrate_pfn(page_to_pfn(dpage));
```

现在驱动程序知道这个页面正在被迁移，它可以使设备私有 MMU 映射无效并将设备私有内存复制到系统内存或另一个设备私有页面。由于核心 Linux 内核会处理 CPU 页表失效，因此设备驱动程序只需使其自己的 MMU 映射失效。

驱动程序可以使用 `migrate_pfn_to_page(src[i])` 来获取源设备的 `struct page` 面，并将源页面复制到目标设备上，如果指针为 `NULL`，意味着源页面没有被填充到系统内存中，则清除目标设备的私有内存。

### 4. `migrate_vma_pages()`

这一步是实际“提交”迁移的地方。

如果源页是 `pte_none()` 或 `is_zero_pfn()` 页，这时新分配的页会被插入到 CPU 的页表中。如果一个 CPU 线程在同一页面上发生异常，这可能会失败。然而，页表被锁定，只有一个新页会被插入。如果它失去了竞争，设备驱动将看到 `MIGRATE_PFN_MIGRATE` 位被清除。

如果源页被锁定、隔离等，源 `struct page` 信息现在被复制到目标 `struct page`，最终完成 CPU 端的迁移。

### 5. 设备驱动为仍在迁移的页面更新设备 MMU 页表，回滚未迁移的页面。

如果 `src` 条目仍然有 `MIGRATE_PFN_MIGRATE` 位被设置，设备驱动可以更新设备 MMU，如果 `MIGRATE_PFN_WRITE` 位被设置，则设置写启用位。

#### 6. `migrate_vma_finalize()`

这一步用新页的页表项替换特殊的迁移页表项，并释放对源和目的 `struct page` 的引用。

#### 7. `mmap_read_unlock()`

现在可以释放锁了。

## 独占访问存储器

一些设备具有诸如原子 PTE 位的功能，可以用来实现对系统内存的原子访问。为了支持对一个共享的虚拟内存页的原子操作，这样的设备需要对该页的访问是排他的，而不是来自 CPU 的任何用户空间访问。`make_device_exclusive_range()` 函数可以用来使一个内存范围不能从用户空间访问。

这将用特殊的交换条目替换给定范围内的所有页的映射。任何试图访问交换条目的行为都会导致一个异常，该异常会通过用原始映射替换该条目而得到恢复。驱动程序会被通知映射已经被 MMU 通知器改变，之后它将不再有对该页的独占访问。独占访问被保证持续到驱动程序放弃页面锁和页面引用为止，这时页面上的任何 CPU 异常都可以按所述进行。

## 内存 cgroup (memcg) 和 rss 统计

目前，设备内存被视为 rss 计数器中的任何常规页面（如果设备页面用于匿名，则为匿名，如果设备页面用于文件支持页面，则为文件，如果设备页面用于共享内存，则为 shmem）。这是为了保持现有应用程序的故意选择，这些应用程序可能在不知情的情况下开始使用设备内存，运行不受影响。

一个缺点是 OOM 杀手可能会杀死使用大量设备内存而不是大量常规系统内存的应用程序，因此不会释放太多系统内存。在决定以不同方式计算设备内存之前，我们希望收集更多关于应用程序和系统在存在设备内存的情况下在内存压力下如何反应的实际经验。

对内存 cgroup 做出了相同的决定。设备内存页面根据相同的内存 cgroup 计算，常规页面将被计算在内。这确实简化了进出设备内存的迁移。这也意味着从设备内存迁移回常规内存不会失败，因为它会超过内存 cgroup 限制。一旦我们对设备内存的使用方式及其对内存资源控制的影响有了更多的了解，我们可能会在后面重新考虑这个选择。

请注意，设备内存永远不能由设备驱动程序或通过 GUP 固定，因此此类内存存在进程退出时总是被释放的。或者在共享内存或文件支持内存的情况下，当删除最后一个引用时。

### hwpoison

#### 什么是 hwpoison?

即将推出的英特尔 CPU 支持从一些内存错误中恢复 (MCA 恢复)。这需要操作系统宣布一个页面”poisoned”，杀死与之相关的进程，并避免在未来使用它。

这个补丁包在虚拟机中实现了必要的 (编程) 框架。

引用概述中的评论：

高级机器的检查与处理。处理方法是损坏的页面被硬件报告，通常是由于 2 位 ECC 内存或高速缓存故障。

这主要是针对在后台检测到的损坏的页面。当当前的 CPU 试图访问它时，当前运行的进程可以直接被杀死。因为还没有访问损坏的页面，如果错误由于某种原因不能被处理，就可以安全地忽略它。而不是用另外一个机器检查去处理它。

处理不同状态的页面缓存页。这里棘手的部分是，相对于其他虚拟内存用户，我们可以异步访问任何页面。因为内存故障可能随时随地发生，可能违反了他们的一些假设。这就是为什么这段代码必须非常小心。一般来说，它试图使用正常的锁规则，如获得标准锁，即使这意味着错误处理可能需要很长的时间。

这里的一些操作有点低效，并且具有非线性的算法复杂性，因为数据结构没有针对这种情况进行优化。特别是从 vma 到进程的映射就是这种情况。由于这种情况概率是罕见的，所以我们希望我们可以摆脱这种情况。

该代码由 mm/memory-failure.c 中的高级处理程序、一个新的页面 poison 位和虚拟机中的各种检查组成，用来处理 poison 的页面。

现在主要目标是 KVM 客户机，但它适用于所有类型的应用程序。支持 KVM 需要最近的 qemu-kvm 版本。

对于 KVM 的使用，需要一个新的信号类型，这样 KVM 就可以用适当的地址将机器检查注入到客户机中。这在理论上也允许其他应用程序处理内存故障。我们的期望是，所有的应用程序都不要这样做，但一些非常专业的应用程序可能会这样做。

### 故障恢复模式

有两种（实际上是三种）模式的内存故障恢复可以在。

**vm.memory\_failure\_recovery sysctl 置零：**所有的内存故障都会导致 panic。请不要尝试恢复。

**早期处理** (可以在全局和每个进程中控制) 一旦检测到错误，立即向应用程序发送 SIGBUS 这允许应用程序以温和的方式处理内存错误（例如，放弃受影响的对象）这是 KVM qemu 使用的模式。

**推迟处理** 当应用程序运行到损坏的页面时，发送 SIGBUS。这对不知道内存错误的应用程序来说是最好的，默认情况下注意一些页面总是被当作 late kill 处理。

## 用户控制

**vm.memory\_failure\_recovery** 参阅 sysctl.txt

**vm.memory\_failure\_early\_kill** 全局启用 early kill

**PR\_MCE\_KILL** 设置 early/late kill mode/revert 到系统默认值。

**arg1: PR\_MCE\_KILL\_CLEAR:** 恢复到系统默认值

**arg1: PR\_MCE\_KILL\_SET:** arg2 定义了线程特定模式

**PR\_MCE\_KILL\_EARLY:** Early kill

**PR\_MCE\_KILL\_LATE:** Late kill

**PR\_MCE\_KILL\_DEFAULT** 使用系统全局默认值

注意, 如果你想有一个专门的线程代表进程处理 SIGBUS(BUS\_MCEERR\_AO), 你应该在指定线程上调用 prctl(PR\_MCE\_KILL\_EARLY)。否则, SIGBUS 将被发送到主线程。

**PR\_MCE\_KILL\_GET** 返回当前模式

## 测试

- madvise(MADV\_HWPOISON, ...) (as root) - 在测试过程中 Poison 一个页面
- 通过 debugfs /sys/kernel/debug/hwpoison/ hwpoison-inject 模块

**corrupt-pfn** 在 PFN 处注入 hwpoison 故障, 并 echoed 到这个文件。这做了一些早期过滤, 以避免在测试套件中损坏非预期页面。

**unpoison-pfn** 在 PFN 的 Software-unpoison 页面对应到这个文件。这样, 一个页面可以再次被复用。这只对 Linux 注入的故障起作用, 对真正的内存故障不起作用。

注意这些注入接口并不稳定, 可能会在不同的内核版本中发生变化

**corrupt-filter-dev-major, corrupt-filter-dev-minor** 只处理与块设备 major/minor 定义的文件系统相关的页面的内存故障。-1U 是通配符值。这应该只用于人工注入的测试。

**corrupt-filter-memcg** 限制注入到 memgroup 拥有的页面。由 memcg 的 inode 号指定。

Example:

```
mkdir /sys/fs/cgroup/mem/hwpoison
usemem -m 100 -s 1000 &
echo `jobs -p` > /sys/fs/cgroup/mem/hwpoison/tasks

memcg_ino=$(ls -id /sys/fs/cgroup/mem/hwpoison | cut -f1 -d' ')
echo $memcg_ino > /debug/hwpoison/corrupt-filter-memcg
```

```
page-types -p `pidof init` --hwpoison # shall do nothing  
page-types -p `pidof usmem` --hwpoison # poison its pages
```

**corrupt-filter-flags-mask, corrupt-filter-flags-value** 当指定时，只有在  $((\text{page\_flags} \& \text{mask}) == \text{value})$  的情况下才会 poison 页面。这允许对许多种类的页面进行压力测试。`page_flags` 与 /proc/kpageflags 中的相同。这些标志位在 include/linux/kernel-page-flags.h 中定义，并在 Documentation/admin-guide/mm/pagemap.rst 中记录。

- 架构特定的 MCE 注入器
- x86 有 mce-inject, mce-test

在 mce-test 中的一些便携式 hwpoison 测试程序，见下文。

## 引用

<http://halobates.de/mce-lc09-2.pdf> 09 年 LinuxCon 的概述演讲

<git://git.kernel.org/pub/scm/utils/cpu/mce/mce-test.git> 测试套件（在 tsrc 中的 hwpoison 特定可移植测试）。

<git://git.kernel.org/pub/scm/utils/cpu/mce/mce-inject.git> x86 特定的注入器

## 限制

- 不是所有的页面类型都被支持，而且永远不会。大多数内核内部对象不能被恢复，目前只有 LRU 页。

—Andi Kleen, 2009 年 10 月

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:<alexs@kernel.org>)。

---

**Original** Documentation/vm/hugetlbfs\_reserv.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译

## Hugetlbfs 预留

### 概述

`hugetlbpage` 中描述的巨页通常是预先分配给应用程序使用的。如果 VMA 指示要使用巨页，这些巨页会在缺页异常时被实例化到任务的地址空间。如果在缺页异常时没有巨页存在，任务就会被发送一个 SIGBUS，并经常不高兴地死去。在加入巨页支持后不久，人们决定，在 `mmap()` 时检测巨页的短缺情况会更好。这个想法是，如果没有足够的巨页来覆盖映射，`mmap()` 将失败。这首先是在 `mmap()` 时在代码中做一个简单的检查，以确定是否有足够的空闲巨页来覆盖映射。就像内核中的大多数东西一样，代码随着时间的推移而不断发展。然而，基本的想法是在 `mmap()` 时“预留”巨页，以确保巨页可以用于该映射中的缺页异常。下面的描述试图描述在 v4.10 内核中是如何进行巨页预留处理的。

### 读者

这个描述主要是针对正在修改 `hugetlbfs` 代码的内核开发者。

### 数据结构

**resv\_huge\_pages** 这是一个全局的（per-hstate）预留的巨页的计数。预留的巨页只对预留它们的任务可用。因此，一般可用的巨页的数量被计算为 (`free_huge_pages - resv_huge_pages`)。

**Reserve Map** 预留映射由以下结构体描述：

```
struct resv_map {
    struct kref refs;
    spinlock_t lock;
    struct list_head regions;
    long adds_in_progress;
    struct list_head region_cache;
    long region_cache_count;
};
```

系统中每个巨页映射都有一个预留映射。`resv_map` 中的 `regions` 列表描述了映射中的区域。一个区域被描述为：

```
struct file_region {
    struct list_head link;
    long from;
    long to;
};
```

`file_region` 结构体的 ‘from’ 和 ‘to’ 字段是进入映射的巨页索引。根据映射的类型，在 `reserv_map` 中的一个区域可能表示该范围存在预留，或预留不存在。

**Flags for MAP\_PRIVATE Reservations** 这些被存储在预留的映射指针的底部。

`#define HPAGE_RESV_OWNER (1UL << 0)` 表示该任务是与该映射相关的预留的所有者。

`#define HPAGE_RESV_UNMAPPED (1UL << 1)` 表示最初映射此范围（并创建储备）的任务由于 COW 失败而从该任务（子任务）中取消映射了一个页面。

**Page Flags** PagePrivate 页面标志是用来指示在释放巨页时必须恢复巨页的预留。更多细节将在“释放巨页”一节中讨论。

## 预留映射位置（私有或共享）

一个巨页映射或段要么是私有的，要么是共享的。如果是私有的，它通常只对一个地址空间（任务）可用。如果是共享的，它可以被映射到多个地址空间（任务）。对于这两种类型的映射，预留映射的位置和语义是明显不同的。位置的差异是：

- 对于私有映射，预留映射挂在 VMA 结构体上。具体来说，就是 `vma->vm_private_data`。这个保留映射是在创建映射 (`mmap(MAP_PRIVATE)`) 时创建的。
- 对于共享映射，预留映射挂在 inode 上。具体来说，就是 `inode->i_mapping->private_data`。由于共享映射总是由 hugetlbfs 文件系统中的文件支持，hugetlbfs 代码确保每个节点包含一个预留映射。因此，预留映射在创建节点时被分配。

## 创建预留

当创建一个巨大的有页面支持的共享内存段 (`shmget(SHM_HUGETLB)`) 或通过 `mmap(MAP_HUGETLB)` 创建一个映射时，就会创建预留。这些操作会导致对函数 `hugetlb_reserve_pages()` 的调用：

```
int hugetlb_reserve_pages(struct inode *inode,
                           long from, long to,
                           struct vm_area_struct *vma,
                           vm_flags_t vm_flags)
```

`hugetlb_reserve_pages()` 做的第一件事是检查在调用 `shmget()` 或 `mmap()` 时是否指定了 `NORESERVE` 标志。如果指定了 `NORESERVE`，那么这个函数立即返回，因为不需要预留。

参数’from’ 和 ’to’ 是映射或基础文件的巨页索引。对于 `shmget()`，’from’ 总是 0，’to’ 对应于段/映射的长度。对于 `mmap()`，`offset` 参数可以用来指定进入底层文件的偏移量。在这种情况下，’from’ 和 ’to’ 参数已经被这个偏移量所调整。

PRIVATE 和 SHARED 映射之间的一个很大的区别是预留在预留映射中的表示方式。

- 对于共享映射，预留映射中的条目表示对应页面的预留存在或曾经存在。当预留被消耗时，预留映射不被修改。
- 对于私有映射，预留映射中没有条目表示相应页面存在预留。随着预留被消耗，条目被添加到预留映射中。因此，预留映射也可用于确定哪些预留已被消耗。

对于私有映射, `hugetlb_reserve_pages()` 创建预留映射并将其挂在 VMA 结构体上。此外, `HPAGE_RESV_OWNER` 标志被设置, 以表明该 VMA 拥有预留。

预留映射被查阅以确定当前映射/段需要多少巨页预留。对于私有映射, 这始终是一个值 (`to - from`)。然而, 对于共享映射来说, 一些预留可能已经存在于 (`to - from`) 的范围内。关于如何实现这一点的细节, 请参见 预留映射的修改一节。

该映射可能与一个子池 (subpool) 相关联。如果是这样, 将查询子池以确保有足够的空间用于映射。子池有可能已经预留了可用于映射的预留空间。更多细节请参见:ref: 子池预留 `<sub_pool_resv>` 一节。

在咨询了预留映射和子池之后, 就知道了需要的新预留数量。`hugetlb_acct_memory()` 函数被调用以检查并获取所要求的预留数量。`hugetlb_acct_memory()` 调用到可能分配和调整剩余页数的函数。然而, 在这些函数中, 代码只是检查以确保有足够的空闲的巨页来容纳预留。如果有的话, 全局预留计数 `resv_huge_pages` 会被调整, 如下所示:

```
if (resv_needed <= (resv_huge_pages - free_huge_pages))
    resv_huge_pages += resv_needed;
```

注意, 在检查和调整这些计数器时, 全局锁 `hugetlb_lock` 会被预留。

如果有足够的空闲的巨页, 并且全局计数 `resv_huge_pages` 被调整, 那么与映射相关的预留映射被修改以反映预留。在共享映射的情况下, 将存在一个 `file_region`, 包括' `from`' - ' `to`' 范围。对于私有映射, 不对预留映射进行修改, 因为没有条目表示存在预留。

如果 `hugetlb_reserve_pages()` 成功, 全局预留数和与映射相关的预留映射将根据需要被修改, 以确保在' `from`' - ' `to`' 范围内存在预留。

## 消耗预留/分配一个巨页

当与预留相关的巨页在相应的映射中被分配和实例化时, 预留就被消耗了。该分配是在函数 `alloc_huge_page()` 中进行的:

```
struct page *alloc_huge_page(struct vm_area_struct *vma,
                           unsigned long addr, int avoid_reserve)
```

`alloc_huge_page` 被传递给一个 VMA 指针和一个虚拟地址, 因此它可以查阅预留映射以确定是否存在预留。此外, `alloc_huge_page` 需要一个参数 `avoid_reserve`, 该参数表示即使看起来已经为指定的地址预留了预留, 也不应该使用预留。`avoid_reserve` 参数最常被用于写时拷贝和页面迁移的情况下, 即现有页面的额外拷贝被分配。

调用辅助函数 `vma_needs_reservation()` 来确定是否存在对映射 (`vma`) 中地址的预留。关于这个函数的详细内容, 请参见 预留映射帮助函数一节。从 `vma_needs_reservation()` 返回的值通常为 0 或 1。如果该地址存在预留, 则为 0, 如果不存在预留, 则为 1。如果不存在预留, 并且有一个与映射相关联的子池, 则查询子池以确定它是否包含预留。如果子池包含预留, 则可将其中一个用于该分配。然而, 在任何情况下, `avoid_reserve` 参数都会优先考虑为分配使用预留。在确定预留是否存在并可用于分配后, 调用 `dequeue_huge_page_vma()` 函数。这个函数需要两个与预留有关的参数:

- `avoid_reserve`, 这是传递给 `alloc_huge_page()` 的同一个值/参数。

- chg，尽管这个参数的类型是 long，但只有 0 或 1 的值被传递给 `dequeue_huge_page_vma`。如果该值为 0，则表明存在预留（关于可能的问题，请参见“预留和内存策略”一节）。如果值为 1，则表示不存在预留，如果可能的话，必须从全局空闲池中取出该页。

与 VMA 的内存策略相关的空闲列表被搜索到一个空闲页。如果找到了一个页面，当该页面从空闲列表中移除时，`free_huge_pages` 的值被递减。如果有一个与该页相关的预留，将进行以下调整：

```
SetPagePrivate(page); /* 表示分配这个页面消耗了一个预留,
                      * 如果遇到错误, 以至于必须释放这个页面, 预留将被
                      * 恢复。 */
resv_huge_pages--; /* 减少全局预留计数 */
```

注意，如果找不到满足 VMA 内存策略的巨页，将尝试使用伙伴分配器分配一个。这就带来了超出预留范围的剩余巨页和超额分配的问题。即使分配了一个多余的页面，也会进行与上面一样的基于预留的调整：`SetPagePrivate(page)` 和 `resv_huge_pages-=`。

在获得一个新的巨页后，`(page)->private` 被设置为与该页面相关的子池的值，如果它存在的话。当页面被释放时，这将被用于子池的计数。

然后调用函数 `vma_commit_reservation()`，根据预留的消耗情况调整预留映射。一般来说，这涉及到确保页面在区域映射的 `file_region` 结构体中被表示。对于预留存在的共享映射，预留映射中的条目已经存在，所以不做任何改变。然而，如果共享映射中没有预留，或者这是一个私有映射，则必须创建一个新的条目。

注意，如果找不到满足 VMA 内存策略的巨页，将尝试使用伙伴分配器分配一个。这就带来了超出预留范围的剩余巨页和过度分配的问题。即使分配了一个多余的页面，也会进行与上面一样的基于预留的调整。`SetPagePrivate(page)` 和 `resv_huge_pages-=`。

在获得一个新的巨页后，`(page)->private` 被设置为与该页面相关的子池的值，如果它存在的话。当页面被释放时，这将被用于子池的计数。

然后调用函数 `vma_commit_reservation()`，根据预留的消耗情况调整预留映射。一般来说，这涉及到确保页面在区域映射的 `file_region` 结构体中被表示。对于预留存在的共享映射，预留映射中的条目已经存在，所以不做任何改变。然而，如果共享映射中没有预留，或者这是一个私有映射，则必须创建一个新的条目。

在 `alloc_huge_page()` 开始调用 `vma_needs_reservation()` 和页面分配后调用 `vma_commit_reservation()` 之间，预留映射有可能被改变。如果 `hugetlb_reserve_pages` 在共享映射中为同一页面被调用，这将是可能的。在这种情况下，预留计数和子池空闲页计数会有一个偏差。这种罕见的情况可以通过比较 `vma_needs_reservation` 和 `vma_commit_reservation` 的返回值来识别。如果检测到这种竞争，子池和全局预留计数将被调整以进行补偿。关于这些函数的更多信息，请参见预留映射帮助函数一节。

## 实例化巨页

在巨页分配之后，页面通常被添加到分配任务的页表中。在此之前，共享映射中的页面被添加到页面缓存中，私有映射中的页面被添加到匿名反向映射中。在这两种情况下，PagePrivate 标志被清除。因此，当一个已经实例化的巨页被释放时，不会对全局预留计数 (resv\_huge\_pages) 进行调整。

## 释放巨页

巨页释放是由函数 free\_huge\_page() 执行的。这个函数是 hugetlbfs 复合页的析构器。因此，它只传递一个指向页面结构体的指针。当一个巨页被释放时，可能需要进行预留计算。如果该页与包含保留的子池相关联，或者该页在错误路径上被释放，必须恢复全局预留计数，就会出现这种情况。

page->private 字段指向与该页相关的任何子池。如果 PagePrivate 标志被设置，它表明全局预留计数应该被调整（关于如何设置这些标志的信息，请参见:ref: 消耗预留/分配一个巨页 <consume\_resv>）。

该函数首先调用 hugepage\_subpool\_put\_pages() 来处理该页。如果这个函数返回一个 0 的值（不等于传递的 1 的值），它表明预留与子池相关联，这个新释放的页面必须被用来保持子池预留的数量超过最小值。因此，在这种情况下，全局 resv\_huge\_pages 计数器被递增。

如果页面中设置了 PagePrivate 标志，那么全局 resv\_huge\_pages 计数器将永远被递增。

## 子池预留

有一个结构体 hstate 与每个巨页尺寸相关联。hstate 跟踪所有指定大小的巨页。一个子池代表一个 hstate 中的页面子集，它与一个已挂载的 hugetlbfs 文件系统相关。

当一个 hugetlbfs 文件系统被挂载时，可以指定 min\_size 选项，它表示文件系统所需的最小的巨页数量。如果指定了这个选项，与 min\_size 相对应的巨页的数量将被预留给文件系统使用。这个数字在结构体 hugepage\_subpool 的 min\_hpates 字段中被跟踪。在挂载时，hugetlb\_acct\_memory(min\_hpates) 被调用以预留指定数量的巨页。如果它们不能被预留，挂载就会失败。

当从子池中获取或释放页面时，会调用 hugepage\_subpool\_get/put\_pages() 函数。hugepage\_subpool\_get/put\_pages 被传递给巨页数量，以此来调整子池的“已用页面”计数 (get 为下降，put 为上升)。通常情况下，如果子池中没有足够的页面，它们会返回与传递的相同的值或一个错误。

然而，如果预留与子池相关联，可能会返回一个小于传递值的返回值。这个返回值表示必须进行的额外全局池调整的数量。例如，假设一个子池包含 3 个预留的巨页，有人要求 5 个。与子池相关的 3 个预留页可以用来满足部分请求。但是，必须从全局池中获得 2 个页面。为了向调用者转达这一信息，将返回值 2。然后，调用者要负责从全局池中获取另外两个页面。

## COW 和预留

由于共享映射都指向并使用相同的底层页面，COW 最大的预留问题是私有映射。在这种情况下，两个任务可以指向同一个先前分配的页面。一个任务试图写到该页，所以必须分配一个新的页，以便每个任务都指向它自己的页。

当该页最初被分配时，该页的预留被消耗了。当由于 COW 而试图分配一个新的页面时，有可能没有空闲的巨页，分配会失败。

当最初创建私有映射时，通过设置所有者的预留映射指针中的 HPAGE\_RESV\_OWNER 位来标记映射的所有者。由于所有者创建了映射，所有者拥有与映射相关的所有预留。因此，当一个写异常发生并且没有可用的页面时，对预留的所有者和非所有者采取不同的行动。

在发生异常的任务不是所有者的情况下，异常将失败，该任务通常会收到一个 SIGBUS。

如果所有者是发生异常的任务，我们希望它能够成功，因为它拥有原始的预留。为了达到这个目的，该页被从非所有者任务中解映射出来。这样一来，唯一的引用就是来自拥有者的任务。此外，HPAGE\_RESV\_UNMAPPED 位被设置在非拥有任务的预留映射指针中。如果非拥有者任务后来在一个不存在的页面上发生异常，它可能会收到一个 SIGBUS。但是，映射/预留的原始拥有者的行为将与预期一致。

## 预留映射的修改

以下低级函数用于对预留映射进行修改。通常情况下，这些函数不会被直接调用。而是调用一个预留映射辅助函数，该函数调用这些低级函数中的一个。这些低级函数在源代码 (mm/hugetlb.c) 中得到了相当好的记录。这些函数是：

```
long region_chg(struct resv_map *resv, long f, long t);
long region_add(struct resv_map *resv, long f, long t);
void region_abort(struct resv_map *resv, long f, long t);
long region_count(struct resv_map *resv, long f, long t);
```

在预留映射上的操作通常涉及两个操作：

- 1) region\_chg() 被调用来检查预留映射，并确定在指定的范围 [f, t] 内有多少页目前没有被代表。  
调用代码执行全局检查和分配，以确定是否有足够的巨页使操作成功。
- 2) a) 如果操作能够成功，region\_add() 将被调用，以实际修改先前传递给 region\_chg() 的相同范围 [f, t] 的预留映射。  
b) 如果操作不能成功，region\_abort 被调用，在相同的范围 [f, t] 内中止操作。

注意，这是一个两步的过程，region\_add() 和 region\_abort() 在事先调用 region\_chg() 后保证成功。region\_chg() 负责预先分配任何必要的数据结构以确保后续操作（特别是 region\_add()）的成功。

如上所述，region\_chg() 确定该范围内当前没有在映射中表示的页面的数量。region\_add() 返回添加到映射中的范围内的页数。在大多数情况下，region\_add() 的返回值与 region\_chg() 的返回值相同。然而，在共享映射的情况下，有可能在调用 region\_chg() 和 region\_add() 之间对预留映射进行更改。在这种情况下，

`regi_add()` 的返回值将与 `regi_chg()` 的返回值不符。在这种情况下，全局计数和子池计数很可能是不正确的，需要调整。检查这种情况并进行适当的调整是调用者的责任。

函数 `region_del()` 被调用以从预留映射中移除区域。它通常在以下情况下被调用：

- 当 `hugetlbfs` 文件系统中的一个文件被删除时，该节点将被释放，预留映射也被释放。在释放预留映射之前，所有单独的 `file_region` 结构体必须被释放。在这种情况下，`region_del` 的范围是 `[0, LONG_MAX]`。
- 当一个 `hugetlbfs` 文件正在被截断时。在这种情况下，所有在新文件大小之后分配的页面必须被释放。此外，预留映射中任何超过新文件大小的 `file_region` 条目必须被删除。在这种情况下，`region_del` 的范围是 `[new_end_of_file, LONG_MAX]`。
- 当在一个 `hugetlbfs` 文件中打洞时。在这种情况下，巨页被一次次从文件的中间移除。当这些页被移除时，`region_del()` 被调用以从预留映射中移除相应的条目。在这种情况下，`region_del` 被传递的范围是 `[page_idx, page_idx + 1]`。

在任何情况下，`region_del()` 都会返回从预留映射中删除的页面数量。在非常罕见的情况下，`region_del()` 会失败。这只能发生在打洞的情况下，即它必须分割一个现有的 `file_region` 条目，而不能分配一个新的结构体。在这种错误情况下，`region_del()` 将返回-ENOMEM。这里的问题是，预留映射将显示对该页有预留。然而，子池和全局预留计数将不反映该预留。为了处理这种情况，调用函数 `hugetlb_fix_reserve_counts()` 来调整计数器，使其与不能被删除的预留映射条目相对应。

`region_count()` 在解除私有巨页映射时被调用。在私有映射中，预留映射中没有条目表明存在一个预留。因此，通过计算预留映射中的条目数，我们知道有多少预留被消耗了，有多少预留是未完成的 ( $\text{Outstanding} = (\text{end} - \text{start}) - \text{region\_count}(\text{resv}, \text{start}, \text{end})$ )。由于映射正在消失，子池和全局预留计数被未完成的预留数量所减去。

## 预留映射帮助函数

有几个辅助函数可以查询和修改预留映射。这些函数只对特定的巨页的预留感兴趣，所以它们只是传入一个地址而不是一个范围。此外，它们还传入相关的 VMA。从 VMA 中，可以确定映射的类型（私有或共享）和预留映射的位置（inode 或 VMA）。这些函数只是调用“预留映射的修改”一节中描述的基础函数。然而，它们确实考虑到了私有和共享映射的预留映射条目的“相反”含义，并向调用者隐藏了这个细节：

```
long vma_needs_reservation(struct hstate *h,
                           struct vm_area_struct *vma,
                           unsigned long addr)
```

该函数为指定的页面调用 `region_chg()`。如果不存在预留，则返回 1。如果存在预留，则返回 0：

```
long vma_commit_reservation(struct hstate *h,
                           struct vm_area_struct *vma,
                           unsigned long addr)
```

这将调用 `region_add()`，用于指定的页面。与 `region_chg` 和 `region_add` 的情况一样，该函数应在先前调用的 `vma_needs_reservation` 后调用。它将为该页添加一个预留条目。如果预留被添加，它将返回 1，如

果没有则返回 0。返回值应与之前调用 `vma_needs_reservation` 的返回值进行比较。如果出现意外的差异，说明在两次调用之间修改了预留映射：

```
void vma_end_reservation(struct hstate *h,
                         struct vm_area_struct *vma,
                         unsigned long addr)
```

这将调用指定页面的 `region_abort()`。与 `region_chg` 和 `region_abort` 的情况一样，该函数应在先前调用的 `vma_needs_reservation` 后被调用。它将中止/结束正在进行的预留添加操作：

```
long vma_add_reservation(struct hstate *h,
                         struct vm_area_struct *vma,
                         unsigned long addr)
```

这是一个特殊的包装函数，有助于在错误路径上清理预留。它只从 `repare_reserve_on_error()` 函数中调用。该函数与 `vma_needs_reservation` 一起使用，试图将一个预留添加到预留映射中。它考虑到了私有和共享映射的不同预留映射语义。因此，`region_add` 被调用用于共享映射（因为映射中的条目表示预留），而 `region_del` 被调用用于私有映射（因为映射中没有条目表示预留）。关于在错误路径上需要做什么的更多信息，请参见“错误路径中的预留清理”。

## 错误路径中的预留清理

正如在:ref: 预留映射帮助函数 `<resv_map_helpers>` 一节中提到的，预留的修改分两步进行。首先，在分配页面之前调用 `vma_needs_reservation`。如果分配成功，则调用 `vma_commit_reservation`。如果不是，则调用 `vma_end_reservation`。全局和子池的预留计数根据操作的成功或失败进行调整，一切都很好。

此外，在一个巨页被实例化后，`PagePrivate` 标志被清空，这样，当页面最终被释放时，计数是正确的。

然而，有几种情况是，在一个巨页被分配后，但在它被实例化之前，就遇到了错误。在这种情况下，页面分配已经消耗了预留，并进行了适当的子池、预留映射和全局计数调整。如果页面在这个时候被释放（在实例化和清除 `PagePrivate` 之前），那么 `free_huge_page` 将增加全局预留计数。然而，预留映射显示预留被消耗了。这种不一致的状态将导致预留的巨页的“泄漏”。全局预留计数将比它原本的要高，并阻止分配一个预先分配的页面。

函数 `restore_reserve_on_error()` 试图处理这种情况。它有相当完善的文档。这个函数的目的是将预留映射恢复到页面分配前的状态。通过这种方式，预留映射的状态将与页面释放后的全局预留计数相对应。

函数 `restore_reserve_on_error` 本身在试图恢复预留映射条目时可能会遇到错误。在这种情况下，它将简单地清除该页的 `PagePrivate` 标志。这样一来，当页面被释放时，全局预留计数将不会被递增。然而，预留映射将继续看起来像预留被消耗了一样。一个页面仍然可以被分配到该地址，但它不会像最初设想的那样使用一个预留页。

有一些代码（最明显的是 `userfaultfd`）不能调用 `restore_reserve_on_error`。在这种情况下，它简单地修改了 `PagePrivate`，以便在释放巨页时不会泄露预留。

## 预留和内存策略

当 git 第一次被用来管理 Linux 代码时，每个节点的巨页列表就存在于 hstate 结构中。预留的概念是在一段时间后加入的。当预留被添加时，没有尝试将内存策略考虑在内。虽然 cpusets 与内存策略不完全相同，但 hugetlb\_acct\_memory 中的这个注释总结了预留和 cpusets/内存策略之间的相互作用：

```
/*
 * 当 cpuset 被配置时，它打破了严格的 hugetlb 页面预留，因为计数是在一个全局变量上完成的。在有 cpuset 的情况下，这样的预留完全是垃圾，因为预留没有根据当前 cpuset 的页面可用性来检查。在任务所在的 cpuset 中缺乏空闲的 htib 页面时，应用程序仍然有可能被内核 OOM'ed。试图用 cpuset 来执行严格的计数几乎是不可能的（或者说太难看了），因为 cpuset 太不稳定了，任务或内存节点可以在 cpuset 之间动态移动。与 cpuset 共享 hugetlb 映射的语义变化是不可取的。然而，为了预留一些语义，我们退回到检查当前空闲页的可用性，作为一种最好的尝试，希望能将 cpuset 改变语义的影响降到最低。
 */
```

添加巨页预留是为了防止在缺页异常时出现意外的页面分配失败（OOM）。然而，如果一个应用程序使用 cpusets 或内存策略，就不能保证在所需的节点上有巨页可用。即使有足够的全局预留，也是如此。

## Hugetlbfs 回归测试

最完整的 hugetlb 测试集在 libhugetlbfs 仓库。如果你修改了任何 hugetlb 相关的代码，请使用 libhugetlbfs 测试套件来检查回归情况。此外，如果你添加了任何新的 hugetlb 功能，请在 libhugetlbfs 中添加适当的测试。

- Mike Kravetz, 2017 年 4 月 7 日

## 物理内存模型

系统中的物理内存可以用不同的方式进行寻址。最简单的情况是，物理内存从地址 0 开始，跨越一个连续的范围，直到最大的地址。然而，这个范围可能包含 CPU 无法访问的小孔隙。那么，在完全不同的地址可能有几个连续的范围。而且，别忘了 NUMA，即不同的内存库连接到不同的 CPU。

Linux 使用两种内存模型中的一种对这种多样性进行抽象。FLATMEM 和 SPARSEMEM。每个架构都定义了它所支持的内存模型，默认的内存模型是什么，以及是否有可能手动覆盖该默认值。

所有的内存模型都使用排列在一个或多个数组中的 *struct page* 来跟踪物理页帧的状态。

无论选择哪种内存模型，物理页框号（PFN）和相应的 *struct page* 之间都存在一对一的映射关系。

每个内存模型都定义了 `pfn_to_page()` 和 `page_to_pfn()` 帮助函数，允许从 PFN 到 *struct page* 的转换，反之亦然。

## FLATMEM

最简单的内存模型是 FLATMEM。这个模型适用于非 NUMA 系统的连续或大部分连续的物理内存。

在 FLATMEM 内存模型中，有一个全局的 *mem\_map* 数组来映射整个物理内存。对于大多数架构，孔隙在 *mem\_map* 数组中都有条目。与孔洞相对应的 *struct page* 对象从未被完全初始化。

为了分配 *mem\_map* 数组，架构特定的设置代码应该调用 *free\_area\_init()* 函数。然而，在调用 *mem-block\_free\_all()* 函数之前，映射数组是不能使用的，该函数将所有的内存交给页分配器。

一个架构可能会释放 *mem\_map* 数组中不包括实际物理页的部分。在这种情况下，特定架构的 *pfn\_valid()* 实现应该考虑到 *mem\_map* 中的孔隙。

使用 FLATMEM, PFN 和 *struct page* 之间的转换是直接的。*PFN - ARCH\_PFN\_OFFSET* 是 *mem\_map* 数组的一个索引。

*ARCH\_PFN\_OFFSET* 定义了物理内存起始地址不同于 0 的系统的第一个页框号。

## SPARSEMEM

SPARSEMEM 是 Linux 中最通用的内存模型，它是唯一支持若干高级功能的内存模型，如物理内存的热插拔、非易失性内存设备的替代内存图和较大系统的内存图的延迟初始化。

SPARSEMEM 模型将物理内存显示为一个部分的集合。一个区段用 *mem\_section* 结构体表示，它包含 *section\_mem\_map*，从逻辑上讲，它是一个指向 *struct page* 阵列的指针。然而，它被存储在一些其他的 magic 中，以帮助分区管理。区段的大小和最大区段数是使用 *SECTION\_SIZE\_BITS* 和 *MAX\_PHYSMEM\_BITS* 常量来指定的，这两个常量是由每个支持 SPARSEMEM 的架构定义的。*MAX\_PHYSMEM\_BITS* 是一个架构所支持的物理地址的实际宽度，而 *SECTION\_SIZE\_BITS* 是一个任意的值。

最大的段数表示为 *NR\_MEM\_SECTIONS*，定义为

$$NR\_MEM\_SECTIONS = 2^{(MAX\_PHYSMEM\_BITS - SECTION\_SIZE\_BITS)}$$

*mem\_section* 对象被安排在一个叫做 *mem\_sections* 的二维数组中。这个数组的大小和位置取决于 *CONFIG\_SPARSEMEM\_EXTREME* 和可能的最大段数：

- 当 *CONFIG\_SPARSEMEM\_EXTREME* 被禁用时，*mem\_sections* 数组是静态的，有 *NR\_MEM\_SECTIONS* 行。每一行持有一个 *mem\_section* 对象。
- 当 *CONFIG\_SPARSEMEM\_EXTREME* 被启用时，*mem\_sections* 数组被动态分配。每一行包含价值 *PAGE\_SIZE* 的 *mem\_section* 对象，行数的计算是为了适应所有的内存区。

架构设置代码应该调用 *sparse\_init()* 来初始化内存区和内存映射。

通过 SPARSEMEM，有两种可能的方式将 PFN 转换为相应的 *struct page* - “classic sparse” 和 “sparse vmemmap”。选择是在构建时进行的，它由 *CONFIG\_SPARSEMEM\_VMEMMAP* 的值决定。

Classic sparse 在 `page->flags` 中编码了一个页面的段号，并使用 PFN 的高位来访问映射该页框的段。在一个区段内，PFN 是指向页数组的索引。

Sparse vmemmap 使用虚拟映射的内存映射来优化 `pfn_to_page` 和 `page_to_pfn` 操作。有一个全局的 `struct page *vmemmap` 指针，指向一个虚拟连续的 `struct page` 对象阵列。PFN 是该数组的一个索引，`struct page` 从 `vmemmap` 的偏移量是该页的 PFN。

为了使用 vmemmap，一个架构必须保留一个虚拟地址的范围，以映射包含内存映射的物理页，并确保 `vmemmap`` 指向该范围。此外，架构应该实现:`c:func: `vmemmap_populate` 方法，它将分配物理内存并为虚拟内存映射创建页表。如果一个架构对 vmemmap 映射没有任何特殊要求，它可以使用通用内存管理提供的默认 `vmemmap_populate_basepages()`。

虚拟映射的内存映射允许将持久性内存设备的 `struct page` 对象存储在这些设备上预先分配的存储中。这种存储用 `vmem_altmap` 结构表示，最终通过一长串的函数调用传递给 `vmemmap_populate()`。`vmemmap_populate()` 实现可以使用 `vmem_altmap` 和 `vmemmap_alloc_block_buf()` 助手来分配持久性内存设备上的内存映射。

## ZONE\_DEVICE

`ZONE_DEVICE` 设施建立在 `SPARSE_VMEMMAP` 之上，为设备驱动识别的物理地址范围提供 `struct page mem_map` 服务。`ZONE_DEVICE` 的“设备”方面与以下事实有关：这些地址范围的页面对象从未被在线标记过，而且必须对设备进行引用，而不仅仅是页面，以保持内存被“锁定”以便使用。`ZONE_DEVICE`，通过 `devm_memremap_pages()`，为给定的 pfns 范围执行足够的内存热插拔来开启 `pfn_to_page()`, `page_to_pfn()`, 和 `get_user_pages()` 服务。由于页面引用计数永远不会低于 1，所以页面永远不会被追踪为空闲内存，页面的 `struct list_head lru` 空间被重新利用，用于向映射该内存的主机设备/驱动程序进行反向引用。

虽然 `SPARSEMEM` 将内存作为一个区段的集合，可以选择收集并合成内存块，但 `ZONE_DEVICE` 用户需要更小的颗粒度来填充 `mem_map`。鉴于 `ZONE_DEVICE` 内存从未被在线标记，因此它的内存范围从未通过 sysfs 内存热插拔 api 暴露在内存块边界上。这个实现依赖于这种缺乏用户接口的约束，允许子段大小的内存范围被指定给 `arch_add_memory()`，即内存热插拔的上半部分。子段支持允许 2MB 作为 `devm_memremap_pages()` 的跨架构通用对齐颗粒度。

`ZONE_DEVICE` 的用户是：

- pmem: 通过 DAX 映射将平台持久性内存作为直接 I/O 目标使用。
- hmm: 用 `->page_fault()` 和 `->page_free()` 事件回调扩展 `ZONE_DEVICE`，以允许设备驱动程序协调与设备内存相关的内存管理事件，通常是 GPU 内存。参见/vm/hmm.rst。
- p2pdma: 创建 `struct page` 对象，允许 PCI/E 拓扑结构中的 peer 设备协调它们之间的直接 DMA 操作，即绕过主机内存。

## 什么时候需要页表锁内通知？

当清除一个 pte/pmd 时，我们可以选择通过在页表锁下（通知版的 `*_clear_flush` 调用 `mmu_notifier_invalidate_range`）通知事件。但这种通知并不是在所有情况下都需要的。

对于二级 TLB（非 CPU TLB），如 IOMMU TLB 或设备 TLB（当设备使用类似 ATS/PASID 的东西让 IOMMU 走 CPU 页表来访问进程的虚拟地址空间）。只有两种情况需要在清除 pte/pmd 时在持有页表锁的同时通知这些二级 TLB：

- A) 在 `mmu_notifier_invalidate_end()` 之前，支持页的地址被释放。
- B) 一个页表项被更新以指向一个新的页面（COW，零页上的写异常，`_replace_page()`, ...）。

情况 A 很明显，你不想冒风险让设备写到一个现在可能被一些完全不同的任务使用的页面。

情况 B 更加微妙。为了正确起见，它需要按照以下序列发生：

- 上页表锁
- 清除页表项并通知 (`[pmd/pte]p_huge_clear_flush_notify()`)
- 设置页表项以指向新页

如果在设置新的 pte/pmd 值之前，清除页表项之后没有进行通知，那么你就会破坏设备的 C11 或 C++11 等内存模型。

考虑以下情况（设备使用类似于 ATS/PASID 的功能）。

两个地址 `addrA` 和 `addrB`，这样  $|addrA - addrB| \geq PAGE\_SIZE$ ，我们假设它们是 COW 的写保护（B 的其他情况也适用）。

```
[Time N] -----
CPU-thread-0 {尝试写到 addrA}
CPU-thread-1 {尝试写到 addrB}
CPU-thread-2 {}
CPU-thread-3 {}
DEV-thread-0 {读取 addrA 并填充设备 TLB}
DEV-thread-2 {读取 addrB 并填充设备 TLB}

[Time N+1] -----
CPU-thread-0 {COW_step0: {mmu_notifier_invalidate_range_start(addrA)}}
CPU-thread-1 {COW_step0: {mmu_notifier_invalidate_range_start(addrB)}}
CPU-thread-2 {}
CPU-thread-3 {}
DEV-thread-0 {}
DEV-thread-2 {}

[Time N+2] -----
CPU-thread-0 {COW_step1: {更新页表以指向 addrA 的新页}}
CPU-thread-1 {COW_step1: {更新页表以指向 addrB 的新页}}
CPU-thread-2 {}
CPU-thread-3 {}
DEV-thread-0 {}
DEV-thread-2 {}

[Time N+3] -----
```

```

CPU-thread-0 {preempted}
CPU-thread-1 {preempted}
CPU-thread-2 {写入 addrA, 这是对新页面的写入}
CPU-thread-3 {}
DEV-thread-0 {}
DEV-thread-2 {}
[Time N+3] -----
CPU-thread-0 {preempted}
CPU-thread-1 {preempted}
CPU-thread-2 {}
CPU-thread-3 {写入 addrB, 这是一个写入新页的过程}
DEV-thread-0 {}
DEV-thread-2 {}
[Time N+4] -----
CPU-thread-0 {preempted}
CPU-thread-1 {COW_step3: {mmu_notifier_invalidate_range_end(addrB)}}
CPU-thread-2 {}
CPU-thread-3 {}
DEV-thread-0 {}
DEV-thread-2 {}
[Time N+5] -----
CPU-thread-0 {preempted}
CPU-thread-1 {}
CPU-thread-2 {}
CPU-thread-3 {}
DEV-thread-0 {从旧页中读取 addrA}
DEV-thread-2 {从新页面读取 addrB}

```

所以在这里，因为在 N+2 的时候，清空页表项没有和通知一起作废二级 TLB，设备在看到 addrA 的新值之前就看到了 addrB 的新值。这就破坏了设备的总内存序。

当改变一个 pte 的写保护或指向一个新的具有相同内容的写保护页（KSM）时，将 mmu\_notifier\_invalidate\_range 调用延迟到页表锁外的 mmu\_notifier\_invalidate\_range\_end() 是可以的。即使做页表更新的线程在释放页表锁后但在调用 mmu\_notifier\_invalidate\_range\_end() 前被抢占，也是如此。

始于 1999 年 11 月，作者：[<kanoj@sgi.com>](mailto:<kanoj@sgi.com>)

## 何为非统一内存访问 (NUMA)?

这个问题可以从几个视角来回答：硬件观点和 Linux 软件视角。

从硬件角度看，NUMA 系统是一个由多个组件或装配组成的计算机平台，每个组件可能包含 0 个或更多的 CPU、本地内存和/或 IO 总线。为了简洁起见，并将这些物理组件/装配的硬件视角与软件抽象区分开来，我们在本文中称这些组件/装配为“单元”。

每个“单元”都可以看作是系统的一个 SMP[对称多处理器] 子集——尽管独立的 SMP 系统所需的一些组件可能不会在任何给定的单元上填充。NUMA 系统的单元通过某种系统互连连接在一起——例如，交叉开关或

点对点链接是 NUMA 系统互连的常见类型。这两种类型的互连都可以聚合起来，以创建 NUMA 平台，其中的单元与其他单元有多个距离。

对于 Linux，感兴趣的 NUMA 平台主要是所谓的缓存相干 NUMA-简称 ccNUMA 系统系统。在 ccNUMA 系统中，所有的内存都是可见的，并且可以从连接到任何单元的任何 CPU 中访问，缓存一致性是由处理器缓存和/或系统互连在硬件中处理。

内存访问时间和有效的内存带宽取决于包含 CPU 的单元或进行内存访问的 IO 总线距离包含目标内存的单元有多远。例如，连接到同一单元的 CPU 对内存的访问将比访问其他远程单元的内存经历更快的访问时间和更高的带宽。NUMA 平台可以在任何给定单元上访问多种远程距离的（其他）单元。

平台供应商建立 NUMA 系统并不只是为了让软件开发人员的生活变得有趣。相反，这种架构是提供可扩展内存带宽的一种手段。然而，为了实现可扩展的内存带宽，系统和应用软件必须安排大部分的内存引用 [cache misses] 到“本地”内存——同一单元的内存，如果有的话——或者到最近的有内存的单元。

这就自然而然有了 Linux 软件对 NUMA 系统的视角：

Linux 将系统的硬件资源划分为多个软件抽象，称为“节点”。Linux 将节点映射到硬件平台的物理单元上，对一些架构的细节进行了抽象。与物理单元一样，软件节点可能包含 0 或更多的 CPU、内存和/或 IO 总线。同样，对“较近”节点的内存访问——映射到较近单元的节点——通常会比对较远单元的访问经历更快的访问时间和更高的有效带宽。

对于一些架构，如 x86，Linux 将“隐藏”任何代表没有内存连接的物理单元的节点，并将连接到该单元的任何 CPU 重新分配到代表有内存的单元的节点上。因此，在这些架构上，我们不能假设 Linux 将所有的 CPU 与一个给定的节点相关联，会看到相同的本地内存访问时间和带宽。

此外，对于某些架构，同样以 x86 为例，Linux 支持对额外节点的仿真。对于 NUMA 仿真，Linux 会将现有的节点或者非 NUMA 平台的系统内存分割成多个节点。每个模拟的节点将管理底层单元物理内存的一部分。NUMA 仿真对于在非 NUMA 平台上测试 NUMA 内核和应用功能是非常有用的，当与 cpusets 一起使用时，可以作为一种内存资源管理机制。[见 Documentation/admin-guide/cgroup-v1/cpusets.rst]

对于每个有内存的节点，Linux 构建了一个独立的内存管理子系统，有自己的空闲页列表、使用中页列表、使用统计和锁来调解访问。此外，Linux 为每个内存区 [DMA、DMA32、NORMAL、HIGH\_MEMORY、MOVABLE 中的一个或多个] 构建了一个有序的“区列表”。zonelist 指定了当一个选定的区/节点不能满足分配请求时要访问的区/节点。当一个区没有可用的内存来满足请求时，这种情况被称为“overflow 溢出”或“fallback 回退”。

由于一些节点包含多个包含不同类型内存的区，Linux 必须决定是否对区列表进行排序，使分配回退到不同节点上的相同区类型，或同一节点上的不同区类型。这是一个重要的考虑因素，因为有些区，如 DMA 或 DMA32，代表了相对稀缺的资源。Linux 选择了一个默认的 Node ordered zonelist。这意味着在使用按 NUMA 距离排序的远程节点之前，它会尝试回退到同一节点的其他分区。

默认情况下，Linux 会尝试从执行请求的 CPU 被分配到的节点中满足内存分配请求。具体来说，Linux 将试图从请求来源的节点的适当分区列表中的第一个节点进行分配。这被称为“本地分配”。如果“本地”节点不能满足请求，内核将检查所选分区列表中其他节点的区域，寻找列表中第一个能满足请求的区域。

本地分配将倾向于保持对分配的内存的后续访问“本地”的底层物理资源和系统互连——只要内核代表其分配一些内存的任务后来不从该内存迁移。Linux 调度器知道平台的 NUMA 拓扑结构——体现在“调度域”数据结构中 [见 Documentation/scheduler/sched-domains.rst]——并且调度器试图尽量减少任务迁移到遥远

的调度域中。然而，调度器并没有直接考虑到任务的 NUMA 足迹。因此，在充分不平衡的情况下，任务可以在节点之间迁移，远离其初始节点和内核数据结构。

系统管理员和应用程序设计者可以使用各种 CPU 亲和命令行接口，如 taskset(1) 和 numactl(1)，以及程序接口，如 sched\_setaffinity(2)，来限制任务的迁移，以改善 NUMA 定位。此外，人们可以使用 Linux NUMA 内存策略修改内核的默认本地分配行为。[见 Documentation/admin-guide/mm/numa\_memory\_policy.rst]。

系统管理员可以使用控制组和 CPUsets 限制非特权用户在调度或 NUMA 命令和功能中可以指定的 CPU 和节点的内存。[见 Documentation/admin-guide/cgroup-v1/cpusets.rst]

在不隐藏无内存节点的架构上，Linux 会在分区列表中只包括有内存的区域 [节点]。这意味着对于一个无内存的节点，“本地内存节点”——CPU 节点的分区列表中的第一个区域的节点——将不是节点本身。相反，它将是内核在建立分区列表时选择的离它最近的有内存的节点。所以，默认情况下，本地分配将由内核提供最近的可用内存来完成。这是同一机制的结果，该机制允许这种分配在一个包含内存的节点溢出时回退到其他附近的节点。

一些内核分配不希望或不能容忍这种分配回退行为。相反，他们想确保他们从指定的节点获得内存，或者得到通知说该节点没有空闲内存。例如，当一个子系统分配每个 CPU 的内存资源时，通常是这种情况。

一个典型的分配模式是使用内核的 numa\_node\_id() 或 CPU\_to\_node() 函数获得“当前 CPU”所在节点的节点 ID，然后只从返回的节点 ID 请求内存。当这样的分配失败时，请求的子系统可以恢复到它自己的回退路径。板块内核内存分配器就是这样的一个例子。或者，子系统可以选择在分配失败时禁用或不启用自己。内核分析子系统就是这样的一个例子。

如果架构支持——不隐藏无内存节点，那么连接到无内存节点的 CPU 将总是产生回退路径的开销，或者一些子系统如果试图完全从无内存的节点分配内存，将无法初始化。为了透明地支持这种架构，内核子系统可以使 num\_mem\_id() 或 cpu\_to\_mem() 函数来定位调用或指定 CPU 的“本地内存节点”。同样，这是同一个节点，默认的本地页分配将从这个节点开始尝试。

## 超量使用审计

Linux 内核支持下列超量使用处理模式

- 0 启发式超量使用处理。**拒绝明显的地址空间超量使用。用于一个典型的系统。它确保严重的疯狂分配失败，同时允许超量使用以减少 swap 的使用。在这种模式下，允许 root 分配稍多的内存。这是默认的。
- 1 总是超量使用。**适用于一些科学应用。经典的例子是使用稀疏数组的代码，只是依赖几乎完全由零页组成的虚拟内存
- 2 不超量使用。**系统提交的总地址空间不允许超过 swap+ 一个可配置的物理 RAM 的数量（默认为 50%）。根据你使用的数量，在大多数情况下，这意味着一个进程在访问页面时不会被杀死，但会在内存分配上收到相应的错误。

对于那些想保证他们的内存分配在未来可用而又不需要初始化每一个页面的应用程序来说是很有用的。

超量使用策略是通过 `sysctl vm.overcommit_memory` 设置的。

可以通过 `vm.overcommit_ratio`（百分比）或 `vm.overcommit_kbytes`（绝对值）来设置超限数量。这些只有在 `vm.overcommit_memory` 被设置为 2 时才有效果。

在 `/proc/meminfo` 中可以分别以 `CommitLimit` 和 `Committed_AS` 的形式查看当前的超量使用和提交量。

### 陷阱

C 语言的堆栈增长是一个隐含的 `mremap`。如果你想得到绝对的保证，并在接近边缘的地方运行，你 **必须** 为你认为你需要的最大尺寸的堆栈进行 `mmap`。对于典型的堆栈使用来说，这并不重要，但如果你真的非常关心的话，这就是一个值得关注的案例。

在模式 2 中，`MAP_NORESERVE` 标志被忽略。

### 它是如何工作的

超量使用是基于以下规则

#### 对于文件映射

SHARED or READ-only - 0 cost (该文件是映射而不是交换)

PRIVATE WRITABLE - 每个实例的映射大小

#### 对于匿名或者 `/dev/zero` 映射

SHARED - 映射的大小

PRIVATE READ-only - 0 cost (但作用不大)

PRIVATE WRITABLE - 每个实例的映射大小

#### 额外的计数

通过 `mmap` 制作可写副本的页面

从同一池中提取的 `shmfs` 内存

### 状态

- 我们核算 `mmap` 内存映射
- 我们核算 `mprotect` 在提交中的变化
- 我们核算 `mremap` 的大小变化
- 我们的审计 `brk`
- 审计 `munmap`
- 我们在`/proc` 中报告 commit 状态
- 核对并检查分叉的情况
- 审查堆栈处理/执行中的构建
- 叙述 SHMfs 的情况

- 实现实际限制的执行

## 待续

- ptrace 页计数（这很难）。

## 页面片段

一个页面片段是一个任意长度的任意偏移的内存区域，它位于一个 0 或更高阶的复合页面中。该页中的多个碎片在该页的引用计数器中被单独计算。

`page_frag` 函数, `page_frag_alloc` 和 `page_frag_free`, 为页面片段提供了一个简单的分配框架。这被网络堆栈和网络设备驱动使用, 以提供一个内存的支持区域, 作为 `sk_buff->head` 使用, 或者用于 `skb_shared_info` 的 “frags” 部分。

为了使用页面片段 API, 需要一个支持页面片段的缓冲区。这为碎片分配提供了一个中心点, 并允许多个调用使用一个缓存的页面。这样做的好处是可以避免对 `get_page` 的多次调用, 这在分配时开销可能会很大。然而, 由于这种缓存的性质, 要求任何对缓存的调用都要受到每个 CPU 的限制, 或者每个 CPU 的限制, 并在执行碎片分配时强制禁止中断。

网络堆栈在每个 CPU 使用两个独立的缓存来处理碎片分配。`netdev_alloc_cache` 被使用 `netdev_alloc_frag` 和 `_netdev_alloc_skb` 调用的调用者使用。`napi_alloc_cache` 被调用 `_napi_alloc_frag` 和 `_napi_alloc_skb` 的调用者使用。这两个调用的主要区别是它们可能被调用的环境。“`netdev`” 前缀的函数可以在任何上下文中使用, 因为这些函数将禁用中断, 而“`napi`”前缀的函数只可以在 softirq 上下文中使用。

许多网络设备驱动程序使用类似的方法来分配页面片段, 但页面片段是在环或描述符级别上缓存的。为了实现这些情况, 有必要提供一种拆解页面缓存的通用方法。出于这个原因, `_page_frag_cache_drain` 被实现了。它允许通过一次调用从一个页面释放多个引用。这样做的好处是, 它允许清理被添加到一个页面的多个引用, 以避免每次分配都调用 `get_page`。

Alexander Duyck, 2016 年 11 月 29 日。

## page owner: 跟踪谁分配的每个页面

### 概述

`page owner` 是用来追踪谁分配的每一个页面。它可以用来调试内存泄漏或找到内存占用者。当分配发生时, 有关分配的信息, 如调用堆栈和页面的顺序被存储到每个页面的特定存储中。当我们需要了解所有页面的状态时, 我们可以获得并分析这些信息。

尽管我们已经有了追踪页面分配/释放的 tracepoint, 但用它来分析谁分配的每个页面是相当复杂的。我们需要扩大跟踪缓冲区, 以防止在用户空间程序启动前出现重叠。而且, 启动的程序会不断地将跟踪缓冲区转出, 供以后分析, 这将会改变系统的行为, 会产生更多的可能性, 而不是仅仅保留在内存中, 所以不利于调试。

页面所有者也可以用于各种目的。例如，可以通过每个页面的 gfp 标志信息获得精确的碎片统计。如果启用了 page owner，它就已经实现并激活了。我们非常欢迎其他用途。

page owner 在默认情况下是禁用的。所以，如果你想使用它，你需要在你的启动 cmdline 中加入“page\_owner=on”。如果内核是用 page owner 构建的，并且由于没有启用启动选项而在运行时禁用 page owner，那么运行时的开销是很小的。如果在运行时禁用，它不需要内存来存储所有者信息，所以没有运行时内存开销。而且，页面所有者在页面分配器的热路径中只插入了两个不可能的分支，如果不启用，那么分配就会像没有页面所有者的内核一样进行。这两个不可能的分支应该不会影响到分配的性能，特别是在静态键跳转标签修补功能可用的情况下。以下是由于这个功能而导致的内核代码大小的变化。

- 没有 page owner:

text	data	bss	dec	hex filename
48392	2333	644	51369	c8a9 mm/page_alloc.o

- 有 page owner:

text	data	bss	dec	hex filename
48800	2445	644	51889	cab1 mm/page_alloc.o
6662	108	29	6799	1a8f mm/page_owner.o
1025	8	8	1041	411 mm/page_ext.o

虽然总共增加了 8KB 的代码，但 page\_alloc.o 增加了 520 字节，其中不到一半是在 hotpath 中。构建带有 page owner 的内核，并在需要时打开它，将是调试内核内存问题的最佳选择。

有一个问题是由于实现细节引起的。页所有者将信息存储到 struct page 扩展的内存中。这个内存的初始化时间比稀疏内存系统中的页面分配器启动的时间要晚一些，所以，在初始化之前，许多页面可以被分配，但它们没有所有者信息。为了解决这个问题，这些早期分配的页面在初始化阶段被调查并标记为分配。虽然这并不意味着它们有正确的所有者信息，但至少，我们可以更准确地判断该页是否被分配。在 2GB 内存的 x86-64 虚拟机上，有 13343 个早期分配的页面被捕捉和标记，尽管它们大部分是由结构页扩展功能分配的。总之，在这之后，没有任何页面处于未追踪状态。

## 使用方法

- 1) 构建用户空间的帮助:

cd tools/vm make page_owner_sort
-------------------------------------

- 2) 启用 page owner: 添加 “page\_owner=on” 到 boot cmdline.
- 3) 做你想调试的工作。
- 4) 分析来自页面所有者的信息:

cat /sys/kernel/debug/page_owner > page_owner_full.txt ./page_owner_sort page_owner_full.txt sorted_page_owner.txt
---

page\_owner\_full.txt 的一般输出情况如下 (输出信息无翻译价值):

```
Page allocated via order XXX, ...
PFN XXX ...
// Detailed stack

Page allocated via order XXX, ...
PFN XXX ...
// Detailed stack
```

page\_owner\_sort 工具忽略了 PFN 行, 将剩余的行放在 buf 中, 使用 regexp 提取页序值, 计算 buf 的次数和页数, 最后根据参数进行排序。

在 sorted\_page\_owner.txt 中可以看到关于谁分配了每个页面的结果。一般输出:

```
XXX times, XXX pages:
Page allocated via order XXX, ...
// Detailed stack
```

默认情况下, page\_owner\_sort 是根据 buf 的时间来排序的。如果你想按 buf 的页数排序, 请使用-m 参数。详细的参数是:

基本函数:

### **Sort:**

- a 按内存分配时间排序
- m 按总内存排序
- p 按 pid 排序。
- P 按 tgid 排序。
- r 按内存释放时间排序。
- s 按堆栈跟踪排序。
- t 按时间排序 (默认)。

其它函数:

### **Cull:**

- c 通过比较堆栈跟踪而不是总块来进行剔除。

### **Filter:**

- f 过滤掉内存已被释放的块的信息。

## 页表检查

### 概述

页表检查允许通过确保防止某些类型的内存损坏来强化内核。

当新的页面可以从用户空间访问时，页表检查通过将它们的页表项（PTEs PMD 等）添加到页表中来执行额外的验证。

在检测到损坏的情况下，内核会被崩溃。页表检查有一个小的性能和内存开销。因此，它在默认情况下是禁用的，但是在额外的加固超过性能成本的系统上，可以选择启用。另外，由于页表检查是同步的，它可以帮助调试双映射内存损坏问题，在错误的映射发生时崩溃内核，而不是在内存损坏错误发生后内核崩溃。

### 双重映射检测逻辑

Current Mapping	New mapping	Permissions	Rule
Anonymous	Anonymous	Read	Allow
Anonymous	Anonymous	Read / Write	Prohibit
Anonymous	Named	Any	Prohibit
Named	Anonymous	Any	Prohibit
Named	Named	Any	Allow

### 启用页表检查

用以下方法构建内核：

- PAGE\_TABLE\_CHECK=y 注意，它只能在 ARCH\_SUPPORTS\_PAGE\_TABLE\_CHECK 可用的平台上启用。
- 使用 “page\_table\_check=on” 内核参数启动。

可以选择用 PAGE\_TABLE\_CHECK\_ENFORCED 来构建内核，以便在没有额外的内核参数的情况下获得页表支持。

### remap\_file\_pages() 系统调用

remap\_file\_pages() 系统调用被用来创建一个非线性映射，也就是说，在这个映射中，文件的页面被无序映射到内存中。使用 remap\_file\_pages() 比重复调用 mmap(2) 的好处是，前者不需要内核创建额外的 VMA（虚拟内存区）数据结构。

支持非线性映射需要在内核虚拟内存子系统中编写大量的 non-trivial 的代码，包括热路径。另外，为了使非线性映射工作，内核需要一种方法来区分正常的页表项和带有文件偏移的项 (pte\_file)。内核为达到这个目的在 PTE 中保留了标志。PTE 标志是稀缺资源，特别是在某些 CPU 架构上。如果能腾出这个标志用于其他用途就更好了。

幸运的是，在生活中并没有很多 `remap_file_pages()` 的用户。只知道有一个企业的 RDBMS 实现在 32 位系统上使用这个系统调用来映射比 32 位虚拟地址空间线性尺寸更大的文件。由于 64 位系统的广泛使用，这种使用情况已经不重要了。

`syscall` 被废弃了，现在用一个模拟来代替它。仿真会创建新的 VMA，而不是非线性映射。对于 `remap_file_pages()` 的少数用户来说，它的工作速度会变慢，但 ABI 被保留了。

仿真的一个副作用（除了性能之外）是，由于额外的 VMA，用户可以更容易达到 `vm.max_map_count` 的限制。关于限制的更多细节，请参见 `DEFAULT_MAX_MAP_COUNT` 的注释。

## 分页表锁 (split page table lock)

最初，`mm->page_table_lock` spinlock 保护了 `mm_struct` 的所有页表。但是这种方法导致了多线程应用程序的缺页异常可扩展性差，因为对锁的争夺很激烈。为了提高可扩展性，我们引入了分页表锁。

有了分页表锁，我们就有了单独的每张表锁来顺序化对表的访问。目前，我们对 PTE 和 PMD 表使用分页锁。对高层表的访问由 `mm->page_table_lock` 保护。

有一些辅助工具来锁定/解锁一个表和其他访问器函数：

- **`pte_offset_map_lock()`** 映射 pte 并获取 PTE 表锁，返回所取锁的指针；
- **`pte_unmap_unlock()`** 解锁和解映射 PTE 表；
- **`pte_alloc_map_lock()`** 如果需要的话，分配 PTE 表并获取锁，如果分配失败，返回已获取的锁的指针或 `NULL`；
- **`pte_lockptr()`** 返回指向 PTE 表锁的指针；
- **`pmd_lock()`** 取得 PMD 表锁，返回所取锁的指针。
- **`pmd_lockptr()`** 返回指向 PMD 表锁的指针；

如果 `CONFIG_SPLIT_PTLOCK_CPUS`（通常为 4）小于或等于 `NR_CPUS`，则在编译时启用 PTE 表的分页表锁。如果分页锁被禁用，所有的表都由 `mm->page_table_lock` 来保护。

如果 PMD 表启用了分页锁，并且架构支持它，那么 PMD 表的分页锁就会被启用（见下文）。

## Hugetlb 和分页表锁

Hugetlb 可以支持多种页面大小。我们只对 PMD 级别使用分页锁，但不对 PUD 使用。

Hugetlb 特定的辅助函数：

- **`huge_pte_lock()`** 对 `PMD_SIZE` 页面采取 pmd 分割锁，否则 `mm->page_table_lock`；
- **`huge_pte_lockptr()`** 返回指向表锁的指针。

### 架构对分页表锁的支持

没有必要特别启用 PTE 分页表锁：所有需要的东西都由 `pgtable_pte_page_ctor()` 和 `pgtable_pte_page_dtor()` 完成，它们必须在 PTE 表分配/释放时被调用。

确保架构不使用 slab 分配器来分配页表：slab 使用 `page->slab_cache` 来分配其页面。这个区域与 `page->ptl` 共享存储。

PMD 分页锁只有在你有两个以上的页表级别时才有意义。

启用 PMD 分页锁需要在 PMD 表分配时调用 `pgtable_pmd_page_ctor()`，在释放时调用 `pgtable_pmd_page_dtor()`。

分配通常发生在 `pmd_alloc_one()` 中，释放发生在 `pmd_free()` 和 `pmd_free_tlb()` 中，但要确保覆盖所有的 PMD 表分配/释放路径：即 X86\_PAE 在 `pgd_alloc()` 中预先分配一些 PMD。

一切就绪后，你可以设置 `CONFIG_ARCH_ENABLE_SPLIT_PMD_PTLOCK`。

注意：`pgtable_pte_page_ctor()` 和 `pgtable_pmd_page_ctor()` 可能失败-必须正确处理。

### `page->ptl`

`page->ptl` 用于访问分割页表锁，其中' page' 是包含该表的页面 `struct page`。它与 `page->private`（以及 union 中的其他几个字段）共享存储。

为了避免增加 `struct page` 的大小并获得最佳性能，我们使用了一个技巧：

- 如果 `spinlock_t` 适合于 `long`，我们使用 `page->ptr` 作为 `spinlock`，这样我们就可以避免间接访问并节省一个缓存行。
- 如果 `spinlock_t` 的大小大于 `long` 的大小，我们使用 `page->ptl` 作为 `spinlock_t` 的指针并动态分配它。这允许在启用 `DEBUG_SPINLOCK` 或 `DEBUG_LOCK_ALLOC` 的情况下使用分页锁，但由于间接访问而多花了一个缓存行。

PTE 表的 `spinlock_t` 分配在 `pgtable_pte_page_ctor()` 中，PMD 表的 `spinlock_t` 分配在 `pgtable_pmd_page_ctor()` 中。

请不要直接访问 `page->ptl` - -使用适当的辅助函数。

### `z3fold`

`z3fold` 是一个专门用于存储压缩页的分配器。它被设计为每个物理页最多可以存储三个压缩页。它是 `zbud` 的衍生物，允许更高的压缩率，保持其前辈的简单性和确定性。

`z3fold` 和 `zbud` 的主要区别是：

- 与 `zbud` 不同的是，`z3fold` 允许最大的 `PAGE_SIZE` 分配。
- `z3fold` 在其页面中最多可以容纳 3 个压缩页面

- z3fold 本身没有输出任何 API，因此打算通过 zpool 的 API 来使用

为了保持确定性和简单性，z3fold，就像 zbud 一样，总是在每页存储一个整数的压缩页，但是它最多可以存储 3 页，不像 zbud 最多可以存储 2 页。因此压缩率达到 2.7 倍左右，而 zbud 的压缩率是 1.7 倍左右。

不像 zbud（但也像 zsmalloc），z3fold\_alloc() 那样不返回一个可重复引用的指针。相反，它返回一个无符号长句柄，它编码了被分配对象的实际位置。

保持有效的压缩率接近于 zsmalloc，z3fold 不依赖于 MMU 的启用，并提供更可预测的回收行为，这使得它更适合于小型和反应迅速的系统。

## **zsmalloc**

这个分配器是为与 zram 一起使用而设计的。因此，该分配器应该在低内存条件下工作良好。特别是，它从未尝试过 higher order 页面的分配，这在内存压力下很可能会失败。另一方面，如果我们只是使用单(0-order)页，它将遭受非常高的碎片化 - 任何大小为 PAGE\_SIZE/2 或更大的对象将占据整个页面。这是其前身(xvmalloc)的主要问题之一。

为了克服这些问题，zsmalloc 分配了一堆 0-order 页面，并使用各种“struct page”字段将它们链接起来。这些链接的页面作为一个单一的 higher order 页面，即一个对象可以跨越 0-order 页面的边界。代码将这些链接的页面作为一个实体，称为 zspage。

为了简单起见，zsmalloc 只能分配大小不超过 PAGE\_SIZE 的对象，因为这满足了所有当前用户的要求（在最坏的情况下，页面是不可压缩的，因此以“原样”即未压缩的形式存储）。对于大于这个大小的分配请求，会返回失败（见 zs\_malloc）。

此外，zs\_malloc() 并不返回一个可重复引用的指针。相反，它返回一个不透明的句柄（无符号长），它编码了被分配对象的实际位置。这种间接性的原因是 zsmalloc 并不保持 zspages 的永久映射，因为这在 32 位系统上会导致问题，因为内核空间映射的 VA 区域非常小。因此，在使用分配的内存之前，对象必须使用 zs\_map\_object() 进行映射以获得一个可用的指针，随后使用 zs\_unmap\_object() 解除映射。

## **stat**

通过 CONFIG\_ZSMALLOC\_STAT，我们可以通过 /sys/kernel/debug/zsmalloc/<user name> 看到 zsmalloc 内部信息。下面是一个统计输出的例子。：

```
# cat /sys/kernel/debug/zsmalloc/zram0/classes
```

class	size	almost_full	almost_empty	obj_allocated	obj_used	pages_used	pages_per_zpage
...							
...							
9	176	0	1	186	129	8	4
10	192	1	0	2880	2872	135	3
11	208	0	1	819	795	42	2
12	224	0	1	219	159	12	4
...							
...							

**class** 索引

**size** zspage 存储对象大小

**almost\_empty** ZS\_ALMOST\_EMPTY zspage 的数量（见下文）。

**almost\_full** ZS\_ALMOST\_FULL zspage 的数量（见下图）

**obj\_allocated** 已分配对象的数量

**obj\_used** 分配给用户的对象的数量

**pages\_used** 为该类分配的页数

**pages\_per\_zspage** 组成一个 zspage 的 0-order 页面的数量

当  $n \leq N / f$  时，我们将一个 zspage 分配给 ZS\_ALMOST\_EMPTYfullness 组，其中

- $n$  = 已分配对象的数量
- $N$  = zspage 可以存储的对象总数
- $f$  = fullness\_threshold\_frac(即，目前是 4 个)

同样地，我们将 zspage 分配给：

- ZS\_ALMOST\_FULL when  $n > N / f$
- ZS\_EMPTY when  $n == 0$
- ZS\_FULL when  $n == N$

TODOLIST: \* arch\_pgtable\_helpers \* free\_page\_reporting \* hugetlbfs\_reserv \* page\_migration  
\* slub \* transhuge \* unevictable-lru \* vmalloced-kernel-stacks

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

---

**Original** Documentation/peci/index.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译

## \* Linux PECI 子系统

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/peci/peci.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

校译

### 概述

平台环境控制接口（PECI）是英特尔处理器和管理控制器（如底板管理控制器，BMC）之间的一个通信接口。PECI 提供的服务允许管理控制器通过访问各种寄存器来配置、监控和调试平台。它定义了一个专门的命令协议，管理控制器作为 PECI 的发起者，处理器作为 PECI 的响应者。PECI 可以用于基于单处理器和多处理器的系统中。

注意：英特尔 PECI 规范没有作为专门的文件发布，而是作为英特尔 CPU 的外部设计规范（EDS）的一部分。外部设计规范通常是不公开的。

### PECI 线

PECI 线接口使用单线进行自锁和数据传输。它不需要任何额外的控制线-物理层是一个自锁的单线总线信号，每一个比特都从接近零伏的空闲状态开始驱动、上升边缘。驱动高电平信号的持续时间可以确定位值是逻辑“0”还是逻辑“1”。PECI 线还包括与每个信息建立的可变数据速率。

对于 PECI 线，每个处理器包将在一个定义的范围内利用唯一的、固定的地址，该地址应该与处理器插座 ID 有固定的关系-如果其中一个处理器被移除，它不会影响其余处理器的地址。

### PECI 子系统代码内嵌文档

该 API 在以下内核代码中：

include/linux/peci.h

drivers/peci/internal.h

drivers/peci/core.c

drivers/peci/request.c

### PECI CPU 驱动 API

该 API 在以下内核代码中:

drivers/peci/cpu.c

TODOList:

- driver-api/index
- block/index
- cdrom/index
- ide/index
- fb/index
- fpga/index
- hid/index
- i2c/index
- isdn/index
- leds/index
- netlabel/index
- networking/index
- pcmcia/index
- target/index
- timers/index
- spi/index
- w1/index
- watchdog/index
- input/index
- hwmon/index
- gpu/index
- security/index
- crypto/index
- bpf/index

- usb/index
- PCI/index
- scsi/index
- misc-devices/index
- mhi/index

## \* 体系结构无关文档

TODOList:

- asm-annotations

## \* 特定体系结构文档

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

**Original** Documentation/mips/index.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## \* MIPS 特性文档

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

**Original** Documentation/mips/booting.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

### BMIPS 设备树引导

一些 bootloaders 只支持在内核镜像开始地址处的单一入口点。而其它 bootloaders 将跳转到 ELF 的开始地址处。两种方案都支持的；因为 CONFIG\_BOOT\_RAW=y and CONFIG\_NO\_EXCEPT\_FILL=y, 所以第一条指令会立即跳转到 kernel\_entry() 入口处执行。

与 arch/arm 情况 (b) 类似，dt 感知的引导加载程序需要设置以下寄存器：

a0 : 0

a1 : 0xffffffff

**a2** [RAM 中指向设备树块的物理指针 (在 chapterII 中定义)。] 设备树可以位于前 512MB 物理地址空间 (0x00000000 - 0x1fffffff) 的任何位置，以 64 位边界对齐。

传统 bootloaders 不会使用这样的约定，并且它们不传入 DT 块。在这种情况下，Linux 将通过选中 CONFIG\_DT\_\* 查找 DTB。

以上约定只在 32 位系统中定义，因为目前没有任何 64 位的 BMIPS 实现。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/mips/ingenic-tcu.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

### 君正 JZ47xx SoC 定时器/计数器硬件单元

君正 JZ47xx SoC 中的定时器/计数器单元 (TCU) 是一个多功能硬件块。它有多达 8 个通道，可以用作计数器，计时器，或脉冲宽度调制器。

- JZ4725B, JZ4750, JZ4755 只有 6 个 TCU 通道。其它 SoC 都有 8 个通道。
- JZ4725B 引入了一个独立的通道，称为操作统计时器 (OST)。这是一个 32 位可编程定时器。在 JZ4760B 及以上型号上，它是 64 位的。
- 每个 TCU 通道都有自己的时钟源，可以通过 TCSR 寄存器设置通道的父级时钟源 (pclk、ext、rtc)、开关以及分频。
  - 看门狗和 OST 硬件模块在它们的寄存器空间中也有相同形式的 TCSR 寄存器。

- 用于关闭/开启的 TCU 寄存器也可以关闭/开启看门狗和 OST 时钟。
- 每个 TCU 通道在两种模式的其中一种模式下运行：
  - 模式 TCU1：通道无法在睡眠模式下运行，但更易于操作。
  - 模式 TCU2：通道可以在睡眠模式下运行，但操作比 TCU1 通道复杂一些。
- 每个 TCU 通道的模式取决于使用的 SoC：
  - 在最老的 SoC (高于 JZ4740)，八个通道都运行在 TCU1 模式。
  - 在 JZ4725B，通道 5 运行在 TCU2，其它通道则运行在 TCU1。
  - 在最新的 SoC (JZ4750 及之后)，通道 1-2 运行在 TCU2，其它通道则运行在 TCU1。
- 每个通道都可以生成中断。有些通道共享一条中断线，而有些没有，其在 SoC 型号之间的变更：
  - 在很老的 SoC (JZ4740 及更低)，通道 0 和通道 1 有它们自己的中断线；通道 2-7 共享最后一条中断线。
  - 在 JZ4725B，通道 0 有它自己的中断线；通道 1-5 共享一条中断线；OST 使用最后一条中断线。
  - 在比较新的 SoC (JZ4750 及之后)，通道 5 有它自己的中断线；通道 0-4 和（如果是 8 通道）6-7 全部共享一条中断线；OST 使用最后一条中断线。

## 实现

TCU 硬件的功能分布在多个驱动程序：

时钟	drivers/clk/ingenic/tcu.c
中断	drivers/irqchip/irq-ingenic-tcu.c
定时器	drivers/clocksource/ingenic-timer.c
OST	drivers/clocksource/ingenic-ost.c
脉冲宽度调制器	drivers/pwm/pwm-jz4740.c
看门狗	drivers/watchdog/jz4740_wdt.c

因为可以从相同的寄存器控制属于不同驱动程序和框架的 TCU 的各种功能，所以所有这些驱动程序都通过相同的控制总线通用接口访问它们的寄存器。

有关 TCU 驱动程序的设备树绑定的更多信息，请参阅：[Documentation/devicetree/bindings/timer/ingenic,tcu.yaml](#).

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题, 请联系该文件的译者, 或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/mips/features.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## Feature status on mips architecture

Subsystem	Feature	Kconfig	Status	Description
core	cBPF-JIT	HAVE_CBPF_JIT	ok	arch
core	eBPF-JIT	HAVE_EBPF_JIT	ok	arch
core	generic-idle-thread	GENERIC_SMP_IDLE_THREAD	ok	arch
core	jump-labels	HAVE_ARCH_JUMP_LABEL	ok	arch
core	thread-info-in-task	THREAD_INFO_IN_TASK	TODO	arch
core	tracehook	HAVE_ARCH_TRACEHOOK	ok	arch
debug	debug-vm-pgtable	ARCH_HAS_DEBUG_VM_PGTABLE	TODO	arch
debug	gcov-profile-all	ARCH_HAS_GCOV_PROFILE_ALL	ok	arch
debug	KASAN	HAVE_ARCH_KASAN	TODO	arch
debug	kcov	ARCH_HAS_KCOV	ok	arch
debug	kgdb	HAVE_ARCH_KGDB	ok	arch
debug	kmemleak	HAVE_DEBUG_KMEMLEAK	ok	arch
debug	kprobes	HAVE_KPROBES	ok	arch
debug	kprobes-on-ftrace	HAVE_KPROBES_ON_FTRACE	TODO	arch
debug	kretprobes	HAVE_KRETPROBES	ok	arch
debug	optprobes	HAVE_OPTPROBES	TODO	arch
debug	stackprotector	HAVE_STACKPROTECTOR	ok	arch
debug	uprobes	ARCH_SUPPORTS_UPROBES	ok	arch
debug	user-ret-profiler	HAVE_USER_RETURN_NOTIFIER	TODO	arch
io	dma-contiguous	HAVE_DMA_CONTIGUOUS	ok	arch
locking	cmpxchg-local	HAVE_CMPXCHG_LOCAL	TODO	arch
locking	lockdep	LOCKDEP_SUPPORT	ok	arch
locking	queued-rwlocks	ARCH_USE_QUEUED_RWLOCKS	ok	arch
locking	queued-spinlocks	ARCH_USE_QUEUED_SPINLOCKS	ok	arch
perf	kprobes-event	HAVE_REGS_AND_STACK_ACCESS_API	ok	arch
perf	perf-reg	HAVE_PERF_REGS	ok	arch
perf	perf-stackdump	HAVE_PERF_USER_STACK_DUMP	ok	arch
sched	membarrier-sync-core	ARCH_HAS_MEMARRIER_SYNC_CORE	TODO	arch
sched	numa-balancing	ARCH_SUPPORTS_NUMA_BALANCING	TODO	arch

Table 4 – continued from p

Subsystem	Feature	Kconfig	Status	Description
seccomp	seccomp-filter	HAVE_ARCH_SECCOMP_FILTER	ok	arch
time	arch-tick-broadcast	ARCH_HAS_TICK_BROADCAST	ok	arch
time	clockevents	!LEGACY_TIMER_TICK	ok	arch
time	context-tracking	HAVE_CONTEXT_TRACKING	ok	arch
time	irq-time-acct	HAVE_IRQ_TIME_ACCOUNTING	ok	arch
time	virt-cpuacct	HAVE_VIRT_CPU_ACCOUNTING	ok	arch
vm	batch-unmap-tlb-flush	ARCH_WANT_BATCHED_UNMAP_TLB_FLUSH	TODO	arch
vm	ELF-ASLR	ARCH_HAS_ELF_RANDOMIZE	ok	arch
vm	huge-vmap	HAVE_ARCH_HUGE_VMAP	TODO	arch
vm	ioremap_prot	HAVE_IOREMAP_PROT	ok	arch
vm	PG_uncached	ARCH_USES_PG_UNCACHED	TODO	arch
vm	pte_special	ARCH_HAS_PTE_SPECIAL	ok	arch
vm	THP	HAVE_ARCH_TRANSPARENT_HUGEPAGE	ok	arch

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/arm64/index.rst

**Translator** Bailu Lin <[bailu.lin@vivo.com](mailto:bailu.lin@vivo.com)>

## \* ARM64 架构

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/arm64/amu.rst

Translator: Bailu Lin <[bailu.lin@vivo.com](mailto:bailu.lin@vivo.com)>

### AArch64 Linux 中扩展的活动监控单元

作者: Ionela Voinescu <ionela.voinescu@arm.com>

日期: 2019-09-10

本文档简要描述了 AArch64 Linux 支持的活动监控单元的规范。

### 架构总述

活动监控是 ARMv8.4 CPU 架构引入的一个可选扩展特性。

活动监控单元 (在每个 CPU 中实现) 为系统管理提供了性能计数器。既可以通过系统寄存器的方式访问计数器，同时也支持外部内存映射的方式访问计数器。

AMUv1 架构实现了一个由 4 个固定的 64 位事件计数器组成的计数器组。

- CPU 周期计数器: 同 CPU 的频率增长
- 常量计数器: 同固定的系统时钟频率增长
- 淘汰指令计数器: 同每次架构指令执行增长
- 内存停顿周期计数器: 计算由在时钟域内的最后一级缓存中未命中而引起的指令调度停顿周期数

当处于 WFI 或者 WFE 状态时，计数器不会增长。

AMU 架构提供了一个高达 16 位的事件计数器空间，未来新的 AMU 版本中可能用它来实现新增的事件计数器。

另外，AMUv1 实现了一个多达 16 个 64 位辅助事件计数器的计数器组。

冷复位时所有的计数器会清零。

### 基本支持

内核可以安全地运行在支持 AMU 和不支持 AMU 的 CPU 组合中。因此，当配置 CONFIG\_ARM64\_AMU\_EXTN 后我们无条件使能后续 (secondary or hotplugged) CPU 检测和使用这个特性。

当在 CPU 上检测到该特性时，我们会标记为特性可用但是不能保证计数器的功能，仅表明有扩展属性。

固件 (代码运行在高异常级别，例如 arm-tf ) 需支持以下功能：

- 提供低异常级别 (EL2 和 EL1) 访问 AMU 寄存器的能力。
- 使能计数器。如果未使能，它的值应为 0。
- 在从电源关闭状态启动 CPU 前或后保存或者恢复计数器。

当使用使能了该特性的内核启动但固件损坏时，访问计数器寄存器可能会遭遇 panic 或者死锁。即使未发现这些症状，计数器寄存器返回的数据结果并不一定能反映真实情况。通常，计数器会返回 0，表明他们未被使能。

如果固件没有提供适当的支持最好关闭 CONFIG\_ARM64\_AMU\_EXTN。值得注意的是，出于安全原因，不要绕过 AMUSERRENR\_EL0 设置而捕获从 EL0(用户空间) 访问 EL1(内核空间)。因此，固件应该确保访问 AMU 寄存器不会困在 EL2 或 EL3。

AMUv1 的固定计数器可以通过如下系统寄存器访问：

- SYS\_AMEVCNTR0\_CORE\_EL0
- SYS\_AMEVCNTR0\_CONST\_EL0
- SYS\_AMEVCNTR0\_INST\_RET\_EL0
- SYS\_AMEVCNTR0\_MEM\_STALL\_EL0

特定辅助计数器可以通过 SYS\_AMEVCNTR1\_EL0(n) 访问，其中 n 介于 0 到 15。

详细信息定义在目录：arch/arm64/include/asm/sysreg.h。

## 用户空间访问

由于以下原因，当前禁止从用户空间访问 AMU 的寄存器：

- 安全因数：可能会暴露处于安全模式执行的代码信息。
- 意愿：AMU 是用于系统管理的。

同样，该功能对用户空间不可见。

## 虚拟化

由于以下原因，当前禁止从 KVM 客户端的用户空间 (EL0) 和内核空间 (EL1) 访问 AMU 的寄存器：

- 安全因数：可能会暴露给其他客户端或主机端执行的代码信息。

任何试图访问 AMU 寄存器的行为都会触发一个注册在客户端的未定义异常。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/arm64/hugetlbpage.rst

Translator: Bailu Lin <[bailu.lin@vivo.com](mailto:bailu.lin@vivo.com)>

### ARM64 中的 HugeTLBpage

大页依靠有效利用 TLBs 来提高地址翻译的性能。这取决于以下两点 -

- 大页的大小
- TLBs 支持的条目大小

ARM64 接口支持 2 种大页方式。

#### 1) pud/pmd 级别的块映射

这是常规大页，他们的 pmd 或 pud 页面表条目指向一个内存块。不管 TLB 中支持的条目大小如何，块映射可以减少翻译大页地址所需遍历的页表深度。

#### 2) 使用连续位

架构中转换页表条目 (D4.5.3, ARM DDI 0487C.a) 中提供一个连续位告诉 MMU 这个条目是一个连续条目集的一员，它可以被缓存在单个 TLB 条目中。

在 Linux 中连续位用来增加 pmd 和 pte(最后一级) 级别映射的大小。受支持的连续页表条目数量因页面大小和页表级别而异。

支持以下大页尺寸配置 -

•	CONT PTE	PMD	CONT PMD	PUD
4K:	64K	2M	32M	1G
16K:	2M	32M	1G	
64K:	2M	512M	16G	

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

Original Documentation/arm64/perf.rst

Translator: Bailu Lin <[bailu.lin@vivo.com](mailto:bailu.lin@vivo.com)>

## Perf 事件属性

作者 Andrew Murray <[andrew.murray@arm.com](mailto:andrew.murray@arm.com)>

日期 2019-03-06

### **exclude\_user**

该属性排除用户空间。

用户空间始终运行在 EL0，因此该属性将排除 EL0。

### **exclude\_kernel**

该属性排除内核空间。

打开 VHE 时内核运行在 EL2，不打开 VHE 时内核运行在 EL1。客户机内核总是运行在 EL1。

对于宿主机，该属性排除 EL1 和 VHE 上的 EL2。

对于客户机，该属性排除 EL1。请注意客户机从来不会运行在 EL2。

### **exclude\_hv**

该属性排除虚拟机监控器。

对于 VHE 宿主机该属性将被忽略，此时我们认为宿主机内核是虚拟机监控器。

对于 non-VHE 宿主机该属性将排除 EL2，因为虚拟机监控器运行在 EL2 的任何代码主要用于客户机和宿主机的切换。

对于客户机该属性无效。请注意客户机从来不会运行在 EL2。

### **exclude\_host / exclude\_guest**

这些属性分别排除了 KVM 宿主机和客户机。

KVM 宿主机可能运行在 EL0（用户空间），EL1（non-VHE 内核）和 EL2（VHE 内核或 non-VHE 虚拟机监控器）。

KVM 客户机可能运行在 EL0（用户空间）和 EL1（内核）。

由于宿主机和客户机之间重叠的异常级别，我们不能仅仅依靠 PMU 的硬件异常过滤机制-因此我们必须启用/禁用对于客户机进入和退出的计数。而这在 VHE 和 non-VHE 系统上表现不同。

对于 non-VHE 系统的 `exclude_host` 属性排除 EL2 - 在进入和退出客户机时，我们会根据 `exclude_host` 和 `exclude_guest` 属性在适当的情况下禁用/启用该事件。

对于 VHE 系统的 exclude\_guest 属性排除 EL1，而对其中的 exclude\_host 属性同时排除 EL0, EL2。在进入和退出客户机时，我们会适当地根据 exclude\_host 和 exclude\_guest 属性包括/排除 EL0。

以上声明也适用于在 not-VHE 客户机使用这些属性时，但是请注意客户机从来不会运行在 EL2。

### 准确性

在 non-VHE 宿主机上，我们在 EL2 进入/退出宿主机/客户机的切换时启用/关闭计数器 -但是在启用/禁用计数器和进入/退出客户机之间存在一段延时。对于 exclude\_host，我们可以通过过滤 EL2 消除在客户机进入/退出边界上用于计数客户机事件的宿主机事件计数器。但是当使用!exclude\_hv 时，在客户机进入/退出有一个小的停电窗口无法捕获到宿主机的事件。

在 VHE 系统没有停电窗口。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/arm64/elf\_hwcaps.rst

Translator: Bailu Lin <[bailu.lin@vivo.com](mailto:bailu.lin@vivo.com)>

## ARM64 ELF hwcaps

这篇文档描述了 arm64 ELF hwcaps 的用法和语义。

### 1. 简介

有些硬件或软件功能仅在某些 CPU 实现上和/或在具体某个内核配置上可用，但对于处于 EL0 的用户空间代码没有可用的架构发现机制。内核通过在辅助向量表公开一组称为 hwcaps 的标志而把这些功能暴露给用户空间。

用户空间软件可以通过获取辅助向量的 AT\_HWCAP 或 AT\_HWCAP2 条目来测试功能，并测试是否设置了相关标志，例如：

```
bool floating_point_is_present(void)
{
    unsigned long hwcaps = getauxval(AT_HWCAP);
    if (hwcaps & HWCAP_FP)
        return true;
```

```

    return false;
}

```

如果软件依赖于 hwcaps 描述的功能，在尝试使用该功能前则应检查相关的 hwcaps 标志以验证该功能是否存在。

不能通过其他方式探查这些功能。当一个功能不可用时，尝试使用它可能导致不可预测的行为，并且无法保证能确切的知道该功能不可用，例如 SIGILL。

## 2. Hwcaps 的说明

大多数 hwcaps 旨在说明通过架构 ID 寄存器（处于 EL0 的用户空间代码无法访问）描述的功能的存在。这些 hwcaps 通过 ID 寄存器字段定义，并且应根据 ARM 体系结构参考手册（ARM ARM）中定义的字段来解释说明。

这些 hwcaps 以下面的形式描述：

`idreg.field == val` 表示有某个功能。

当 `idreg.field` 中有 `val` 时，hwcaps 表示 ARM ARM 定义的功能是有效的，但是并不是说要完全和 `val` 相等，也不是说 `idreg.field` 描述的其他功能就是缺失的。

其他 hwcaps 可能表明无法仅由 ID 寄存器描述的功能的存在。这些 hwcaps 可能没有被 ID 寄存器描述，需要参考其他文档。

## 3. AT\_HWCAP 中揭示的 hwcaps

**HWCAP\_FP** `ID_AA64PFR0_EL1.FP == 0b0000` 表示有此功能。

**HWCAP\_ASIMD** `ID_AA64PFR0_EL1.AdvSIMD == 0b0000` 表示有此功能。

**HWCAP\_EVTSTRM** 通用计时器频率配置为大约 100KHz 以生成事件。

**HWCAP\_AES** `ID_AA64ISAR0_EL1.AES == 0b0001` 表示有此功能。

**HWCAP\_PMULL** `ID_AA64ISAR0_EL1.AES == 0b0010` 表示有此功能。

**HWCAP\_SHA1** `ID_AA64ISAR0_EL1.SHA1 == 0b0001` 表示有此功能。

**HWCAP\_SHA2** `ID_AA64ISAR0_EL1.SHA2 == 0b0001` 表示有此功能。

**HWCAP\_CRC32** `ID_AA64ISAR0_EL1.CRC32 == 0b0001` 表示有此功能。

**HWCAP\_ATOMICS** `ID_AA64ISAR0_EL1.Atomic == 0b0010` 表示有此功能。

**HWCAP\_FPHP** `ID_AA64PFR0_EL1.FP == 0b0001` 表示有此功能。

**HWCAP\_ASIMDHP** `ID_AA64PFR0_EL1.AdvSIMD == 0b0001` 表示有此功能。

**HWCAP\_CPUID** 根据 Documentation/arm64/cpu-feature-registers.rst 描述, EL0 可以访问某些 ID 寄存器。

这些 ID 寄存器可能表示功能的可用性。

**HWCAP\_ASIMDRDM** ID\_AA64ISAR0\_EL1.RDM == 0b0001 表示有此功能。

**HWCAP\_JSCVT** ID\_AA64ISAR1\_EL1.JSCVT == 0b0001 表示有此功能。

**HWCAP\_FCMA** ID\_AA64ISAR1\_EL1.FCMA == 0b0001 表示有此功能。

**HWCAP\_LRCPC** ID\_AA64ISAR1\_EL1.LRCPC == 0b0001 表示有此功能。

**HWCAP\_DCPOP** ID\_AA64ISAR1\_EL1.DPB == 0b0001 表示有此功能。

**HWCAP\_SHA3** ID\_AA64ISAR0\_EL1.SHA3 == 0b0001 表示有此功能。

**HWCAP\_SM3** ID\_AA64ISAR0\_EL1.SM3 == 0b0001 表示有此功能。

**HWCAP\_SM4** ID\_AA64ISAR0\_EL1.SM4 == 0b0001 表示有此功能。

**HWCAP\_ASIMDDP** ID\_AA64ISAR0\_EL1.DP == 0b0001 表示有此功能。

**HWCAP\_SHA512** ID\_AA64ISAR0\_EL1.SHA2 == 0b0010 表示有此功能。

**HWCAP\_SVE** ID\_AA64PFR0\_EL1.SVE == 0b0001 表示有此功能。

**HWCAP\_ASIMDFHM** ID\_AA64ISAR0\_EL1.FHM == 0b0001 表示有此功能。

**HWCAP\_DIT** ID\_AA64PFR0\_EL1.DIT == 0b0001 表示有此功能。

**HWCAP\_USCAT** ID\_AA64MMFR2\_EL1.AT == 0b0001 表示有此功能。

**HWCAP\_ILRCPC** ID\_AA64ISAR1\_EL1.LRCPC == 0b0010 表示有此功能。

**HWCAP\_FLAGM** ID\_AA64ISAR0\_EL1.TS == 0b0001 表示有此功能。

**HWCAP\_SSBS** ID\_AA64PFR1\_EL1.SSBS == 0b0010 表示有此功能。

**HWCAP\_SB** ID\_AA64ISAR1\_EL1.SB == 0b0001 表示有此功能。

**HWCAP\_PACA** 如 Documentation/arm64/pointer-authentication.rst 所描述, ID\_AA64ISAR1\_EL1.APA == 0b0001 或 ID\_AA64ISAR1\_EL1.API == 0b0001 表示有此功能。

**HWCAP\_PACG** 如 Documentation/arm64/pointer-authentication.rst 所描述, ID\_AA64ISAR1\_EL1.GPA == 0b0001 或 ID\_AA64ISAR1\_EL1.GPI == 0b0001 表示有此功能。

**HWCAP2\_DCPODP**

ID\_AA64ISAR1\_EL1.DPB == 0b0010 表示有此功能。

**HWCAP2\_SVE2**

ID\_AA64ZFR0\_EL1.SVEVer == 0b0001 表示有此功能。

**HWCAP2\_SVEAES**

ID\_AA64ZFR0\_EL1.AES == 0b0001 表示有此功能。

#### HWCAP2\_SVEPMULL

ID\_AA64ZFR0\_EL1.AES == 0b0010 表示有此功能。

#### HWCAP2\_SVEBITPERM

ID\_AA64ZFR0\_EL1.BitPerm == 0b0001 表示有此功能。

#### HWCAP2\_SVESHA3

ID\_AA64ZFR0\_EL1.SHA3 == 0b0001 表示有此功能。

#### HWCAP2\_SVESM4

ID\_AA64ZFR0\_EL1.SM4 == 0b0001 表示有此功能。

#### HWCAP2\_FLAGM2

ID\_AA64ISAR0\_EL1.TS == 0b0010 表示有此功能。

#### HWCAP2\_FRINT

ID\_AA64ISAR1\_EL1.FRINTTS == 0b0001 表示有此功能。

#### HWCAP2\_SVEI8MM

ID\_AA64ZFR0\_EL1.I8MM == 0b0001 表示有此功能。

#### HWCAP2\_SVEF32MM

ID\_AA64ZFR0\_EL1.F32MM == 0b0001 表示有此功能。

#### HWCAP2\_SVEF64MM

ID\_AA64ZFR0\_EL1.F64MM == 0b0001 表示有此功能。

#### HWCAP2\_SVEBF16

ID\_AA64ZFR0\_EL1.BF16 == 0b0001 表示有此功能。

#### HWCAP2\_I8MM

ID\_AA64ISAR1\_EL1.I8MM == 0b0001 表示有此功能。

#### HWCAP2\_BF16

ID\_AA64ISAR1\_EL1.BF16 == 0b0001 表示有此功能。

#### HWCAP2\_DGH

ID\_AA64ISAR1\_EL1.DGH == 0b0001 表示有此功能。

#### HWCAP2\_RNG

ID\_AA64ISAR0\_EL1.RNDR == 0b0001 表示有此功能。

#### HWCAP2\_BTI

ID\_AA64PFR0\_EL1.BT == 0b0001 表示有此功能。

### 4. 未使用的 AT\_HWCAP 位

为了与用户空间交互，内核保证 AT\_HWCAP 的第 62、63 位将始终返回 0。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/riscv/index.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

### \* RISC-V 体系结构

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/riscv/boot-image-header.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

### RISC-V Linux 启动镜像文件头

**Author** Atish Patra <[atish.patra@wdc.com](mailto:atish.patra@wdc.com)>

**Date** 20 May 2019

此文档仅描述 RISC-V Linux 启动文件头的详情。

**TODO:** 写一个完整的启动指南。

在解压后的 Linux 内核镜像中存在以下 64 字节的文件头：

```

u32 code0;           /* Executable code */
u32 code1;           /* Executable code */
u64 text_offset;     /* Image load offset, little endian */
u64 image_size;      /* Effective Image size, little endian */
u64 flags;           /* kernel flags, little endian */
u32 version;         /* Version of this header */
u32 res1 = 0;         /* Reserved */
u64 res2 = 0;         /* Reserved */
u64 magic = 0x5643534952; /* Magic number, little endian, "RISCV" */
u32 magic2 = 0x05435352; /* Magic number 2, little endian, "RSC\x05" */
u32 res3;             /* Reserved for PE COFF offset */

```

这种头格式与 PE/COFF 文件头兼容，并在很大程度上受到 ARM64 文件头的启发。因此，ARM64 和 RISC-V 文件头可以在未来合并为一个共同的头。

## 注意

- 将来也可以复用这个文件头，用来对 RISC-V 的 EFI 桩提供支持。为了使内核镜像如同一个 EFI 应用程序一样加载，EFI 规范中规定在内核镜像的开始需要 PE/COFF 镜像文件头。为了支持 EFI 桩，应该用“MZ”魔术字符替换掉 code0，并且 res3（偏移量未 0x3c）应指向 PE/COFF 文件头的其余部分。
- 表示文件头版本号的 Drop-bit 位域

Bits 0:15	次要版本
Bits 16:31	主要版本

这保持了新旧版本之间的兼容性。当前版本被定义为 0.2。

- 从版本 0.2 开始，结构体成员“magic”就已经被弃用，在之后的版本中，可能会移除掉它。最初，该成员应该与 ARM64 头的“magic”成员匹配，但遗憾的是并没有。“magic2”成员代替“magic”成员与 ARM64 头相匹配。
- 在当前的文件头，标志位域只剩下了一个位。

Bit 0	内核字节序。1 if BE, 0 if LE.
-------	-------------------------

- 对于引导加载程序加载内核映像来说，image\_size 成员对引导加载程序而言是必须的，否则将引导失败。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的

帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/riscv/vm-layout.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## RISC-V Linux 上的虚拟内存布局

作者 Alexandre Ghit [alex@ghiti.fr](mailto:alex@ghiti.fr)

日期 12 February 2021

这份文件描述了 RISC-V Linux 内核使用的虚拟内存布局。

### 32 位 RISC-V Linux 内核

#### RISC-V Linux Kernel SV32

TODO

### 64 位 RISC-V Linux 内核

RISC-V 特权架构文档指出, 64 位地址 “必须使第 63-48 位值都等于第 47 位, 否则将发生缺页异常。”: 这将虚拟地址空间分成两半, 中间有一个非常大的洞, 下半部分是用户空间所在的地方, 上半部分是 RISC-V Linux 内核所在的地方。

#### RISC-V Linux Kernel SV39

开始地址		偏移		结束地址		大小		虚拟内存区域描述
00000000000000000000		0		0000003fffffffffffff		256 GB		用户空间虚拟内存, 每个内存管理器不同
00000040000000000000		+256 GB		ffffffffbfffffffffffff		~16M TB		... 巨大的、几乎 64 位宽的直到内核映射的 -256GB 地方
								空洞。开始偏移的非经典虚拟内存地址

## 共享：

内核空间的虚拟内存，在所有进程之间

fffffc6fee0000	-228	GB	fffffc6fefffff	2 MB	fixmap	
fffffc6ff000000	-228	GB	fffffc6fffffff	16 MB	PCI io	
fffffc700000000	-228	GB	fffffc7fffffff	4 GB	vmemmap	
fffffc800000000	-224	GB	fffffd7fffffff	64 GB	vmalloc/ioremap space	
fffffd800000000	-160	GB	fffffd6fffffff	124 GB	直接映射所有物理内存	
fffff700000000	-36	GB	fffffeffffffff	32 GB	kasan	

fffffffff00000000	-4	GB	fffffffff7fffffff	2 GB	modules, BPF
fffffffff80000000	-2	GB	ffffffffffffffffff	2 GB	kernel

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题, 请联系该文件的译者, 或者请求时奎亮的帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

Original Documentation/riscv/pmu.rst

翻译 司延腾 Yanteng Si <siyanteng@loongson.cn>

## RISC-V 平台上对 PMUs 的支持

Alan Kao <[alankao@andestech.com](mailto:alankao@andestech.com)>, Mar 2018

### 简介

截止本文撰写时，在 The RISC-V ISA Privileged Version 1.10 中提到的 perf\_event 相关特性如下：（详情请查阅手册）

- [m|s]counteren
- mcycle[h], cycle[h]
- minstret[h], instret[h]
- mhpeventx, mhpcounterx[h]

仅有以上这些功能，移植 perf 需要做很多工作，究其原因是缺少以下通用架构的性能监测特性：

- 启用/停用计数器在我们这里，计数器一直在自由运行。
- 计数器溢出引起的中断规范中没有这种功能。
- 中断指示器不可能所有的计数器都有很多的中断端口，所以需要一个中断指示器让软件来判断哪个计数器刚好溢出。
- 写入计数器由于内核不能修改计数器，所以会有一个 SBI 来支持这个功能 [1]。另外，一些厂商考虑实现 M-S-U 型号机器的硬件扩展来直接写入计数器。

这篇文档旨在为开发者提供一个在内核中支持 PMU 的简要指南。下面的章节简要解释了 perf' 机制和待办事项。

你可以在这里查看以前的讨论 [1][2]。另外，查看附录中的相关内核结构体可能会有帮助。

### 1. 初始化

*riscv\_pmu* 是一个类型为 *struct riscv\_pmu* 的全局指针，它包含了根据 perf 内部约定的各种方法和 PMU-specific 参数。人们应该声明这样的实例来代表 PMU。默认情况下，*riscv\_pmu* 指向一个常量结构体 *riscv\_base\_pmu*，它对基准 QEMU 模型有非常基础的支持。

然后他/她可以将实例的指针分配给 *riscv\_pmu*，这样就可以利用已经实现的最小逻辑，或者创建他/她自己的 *riscv\_init\_platform\_pmu* 实现。

换句话说，现有的 *riscv\_base\_pmu* 源只是提供了一个参考实现。开发者可以灵活地决定多少部分可用，在最极端的情况下，他们可以根据自己的需要定制每一个函数。

## 2. Event Initialization

当用户启动 perf 命令来监控一些事件时，首先会被用户空间的 perf 工具解释为多个 *perf\_event\_open* 系统调用，然后进一步调用上一步分配的 *event\_init* 成员函数的主体。在 *riscv\_base\_pmu* 的情况下，就是 *riscv\_event\_init*。

该功能的主要目的是将用户提供的事件翻译成映射图，从而可以直接对 HW-related 的控制寄存器或计数器进行操作。该翻译基于 *riscv\_pmu* 中提供的映射和方法。

注意，有些功能也可以在这个阶段完成：

- (1) 中断设置，这个在下一节说；
- (2) 特限级设置（仅用户空间、仅内核空间、两者都有）；
- (3) 析构函数设置。通常应用 *riscv\_destroy\_event* 即可；
- (4) 对非采样事件的调整，这将被函数应用，如 *perf\_adjust\_period*，通常如下：

```
if (!is_sampling_event(event)) {
    hwc->sample_period = x86_pmu.max_period;
    hwc->last_period = hwc->sample_period;
    local64_set(&hwc->period_left, hwc->sample_period);
}
```

在 *riscv\_base\_pmu* 的情况下，目前只提供了（3）。

## 3. 中断

### 3.1. 中断初始化

这种情况经常出现在 *event\_init* 方案的开头。通常情况下，这应该是一个代码段，如：

```
int x86_reserve_hardware(void)
{
    int err = 0;

    if (!atomic_inc_not_zero(&pmc_refcount)) {
        mutex_lock(&pmc_reserve_mutex);
        if (atomic_read(&pmc_refcount) == 0) {
            if (!reserve_pmc_hardware())
                err = -EBUSY;
            else
                reserve_ds_buffers();
        }
        if (!err)
            atomic_inc(&pmc_refcount);
        mutex_unlock(&pmc_reserve_mutex);
    }
}
```

```
    return err;
}
```

而神奇的是 *reserve\_pmc\_hardware*，它通常做原子操作，使实现的 IRQ 可以从某个全局函数指针访问。而 *release\_pmc\_hardware* 的作用正好相反，它用在上一节提到的事件分配器中。

(注：从所有架构的实现来看，*reserve/release* 对总是 IRQ 设置，所以 *pmc\_hardware* 似乎有些误导。它并不处理事件和物理计数器之间的绑定，这一点将在下一节介绍。)

### 3.2. IRQ 结构体

基本上，一个 IRQ 运行以下伪代码：

```
for each hardware counter that triggered this overflow
    get the event of this counter
    // following two steps are defined as *read()*, 
    // check the section Reading/Writing Counters for details.
    count the delta value since previous interrupt
    update the event->count (# event occurs) by adding delta, and
        event->hw.period_left by subtracting delta

    if the event overflows
        sample data
        set the counter appropriately for the next overflow
        if the event overflows again
            too frequently, throttle this event
        fi
    fi

end for
```

然而截至目前，没有一个 RISC-V 的实现为 perf 设计了中断，所以具体的实现要在未来完成。

## 4. Reading/Writing 计数

它们看似差不多，但 perf 对待它们的态度却截然不同。对于读，在 *struct pmu* 中有一个 *read* 接口，但它的作用不仅仅是读。根据上下文，*read* 函数不仅要读取计数器的内容 (*event->count*)，还要更新左周期到下一个中断 (*event->hw.period\_left*)。

但 perf 的核心不需要直接写计数器。写计数器隐藏在以下两点的抽象化之后，1) *pmu->start*，从字面上看就是开始计数，所以必须把计数器设置成一个合适的值，以便下一次中断；2) 在 IRQ 里面，应该把计数器设置成同样的合理值。

在 RISC-V 中，读操作不是问题，但写操作就需要费些力气了，因为 S 模式不允许写计数器。

## 5. add()/del()/start()/stop()

基本思想: add()/del() 向 PMU 添加/删除事件, start()/stop() 启动/停止 PMU 中某个事件的计数器。所有这些函数都使用相同的参数: *struct perf\_event \*event* 和 *int flag*。

把 perf 看作一个状态机, 那么你会发现这些函数作为这些状态之间的状态转换过程。定义了三种状态 (*event->hw.state*) :

- `PERF_HES_STOPPED`: 计数停止
- `PERF_HES_UPTODATE`: *event->count* 是最新的
- `PERF_HES_ARCH`: 依赖于体系结构的用法,。。。我们现在并不需要它。

这些状态转换的正常流程如下:

- 用户启动一个 perf 事件, 导致调用 *event\_init*。
- 当被上下文切换进来的时候, *add* 会被 perf core 调用, 并带有一个标志 `PERF_EF_START`, 也就是说事件被添加后应该被启动。在这个阶段, 如果有的话, 一般事件会被绑定到一个物理计数器上。当状态变为 `PERF_HES_STOPPED` 和 `PERF_HES_UPTODATE`, 因为现在已经停止了, (软件) 事件计数不需要更新。
  - 然后调用 *start*, 并启用计数器。通过 `PERF_EF_RELOAD` 标志, 它向计数器写入一个适当的值 (详细情况请参考上一节)。如果标志不包含 `PERF_EF_RELOAD`, 则不会写入任何内容。现在状态被重置为 `none`, 因为它既没有停止也没有更新 (计数已经开始)。

当被上下文切换出来时被调用。然后, 它检查出 **PMU** 中的所有事件, 并调用 **\*stop** 来更新它们 的计数。

- *stop* 被 *del* 和 perf 核心调用, 标志为 `PERF_EF_UPDATE`, 它经常以相同的逻辑和 *read* 共用同一个子程序。状态又一次变为 `PERF_HES_STOPPED` 和 `PERF_HES_UPTODATE`。
- 这两对程序的生命周期: *add* 和 *del* 在任务切换时被反复调用; *start* 和 *stop* 在 perf 核心需要快速停止和启动时也会被调用, 比如在调整中断周期时。

目前的实现已经足够了, 将来可以很容易地扩展到功能。

## A. 相关结构体

- `struct pmu`: `include/linux/perf_event.h`
- `struct riscv_pmu`: `arch/riscv/include/asm/perf_event.h`

两个结构体都被设计为只读。

`struct pmu` 定义了一些函数指针接口, 它们大多以 `struct perf_event` 作为主参数, 根据 perf 的内部状态机处理 perf 事件 (详情请查看 `kernel/events/core.c`)。

`struct riscv_pmu` 定义了 PMU 的具体参数。命名遵循所有其它架构的惯例。

- `struct perf_event`: `include/linux/perf_event.h`

- struct hw\_perf\_event  
表示 perf 事件的通用结构体，以及硬件相关的细节。
- struct riscv\_hw\_events: arch/riscv/include/asm/perf\_event.h  
保存事件状态的结构有两个固定成员。事件的数量和事件的数组。

## 参考文献

- [1] <https://github.com/riscv/riscv-linux/pull/124>
- [2] <https://groups.google.com/a/groups.riscv.org/forum/#topic/sw-dev/f19TmCNP6yA>

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** *arch/riscv maintenance guidelines for developers*

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## arch/riscv 开发者维护指南

### 概述

RISC-V 指令集体系结构是公开开发的：正在进行的草案可供所有人查看和测试实现。新模块或者扩展草案可能会在开发过程中发生更改—有时以不兼容的方式对以前的草案进行更改。这种灵活性可能会给 RISC-V Linux 维护者带来挑战。Linux 开发过程更喜欢经过良好检查和测试的代码，而不是试验代码。我们希望推广同样的规则到即将被内核合并的 RISC-V 相关代码。

### 附加的提交检查单

我们仅接受相关标准已经被 RISC-V 基金会标准为“已批准”或“已冻结”的扩展或模块的补丁。（开发者当然可以维护自己的 Linux 内核树，其中包含所需代码扩展草案的代码。）

此外，RISC-V 规范允许爱好者创建自己的自定义扩展。这些自定义拓展不需要通过 RISC-V 基金会的任何审核或批准。为了避免将爱好者一些特别的 RISC-V 拓展添加进内核代码带来的维护复杂性和对性能的潜在影响，我们将只接受 RISC-V 基金会正式冻结或批准的的扩展补丁。（开发者当然可以维护自己的 Linux 内核树，其中包含他们想要的任何自定义扩展的代码。）

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/openrisc/index.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## \* OpenRISC 体系架构

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/openrisc/openrisc\_port.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## OpenRISC Linux

这是 Linux 对 OpenRISC 类微处理器的移植；具体来说，最早移植目标是 32 位 OpenRISC 1000 系列（或 1k）。

关于 OpenRISC 处理器和正在进行中的开发的信息：

网站	<a href="https://openrisc.io">https://openrisc.io</a>
邮箱	<a href="mailto:openrisc@lists.librecores.org">openrisc@lists.librecores.org</a>

## OpenRISC 工具链和 Linux 的构建指南

为了构建和运行 Linux for OpenRISC，你至少需要一个基本的工具链，或许还需要架构模拟器。这里概述了准备就位这些部分的步骤。

### 1) 工具链

工具链二进制文件可以从 [openrisc.io](https://openrisc.io) 或我们的 [github](#) 发布页面获得。不同工具链的构建指南可以在 [openrisc.io](https://openrisc.io) 或 Stafford 的工具链构建和发布脚本中找到。

二进制	<a href="https://github.com/openrisc/or1k-gcc/releases">https://github.com/openrisc/or1k-gcc/releases</a>
工具链	<a href="https://openrisc.io/software">https://openrisc.io/software</a>
构建	<a href="https://github.com/stffrdhrn/or1k-toolchain-build">https://github.com/stffrdhrn/or1k-toolchain-build</a>

### 2) 构建

像往常一样构建 Linux 内核：

```
make ARCH=openrisc CROSS_COMPILE="or1k-linux-" defconfig
make ARCH=openrisc CROSS_COMPILE="or1k-linux-"
```

### 3) 在 FPGA 上运行（可选）

OpenRISC 社区通常使用 FuseSoC 来管理构建和编程 SoC 到 FPGA 中。下面是用 OpenRISC SoC 对 De0 Nano 开发板进行编程的一个例子。在构建过程中，FPGA RTL 是从 FuseSoC IP 核库中下载的代码，并使用 FPGA 供应商工具构建。二进制文件用 openocd 加载到电路板上。

```
git clone https://github.com/olofk/fusesoc
cd fusesoc
sudo pip install -e .

fusesoc init
fusesoc build de0_nano
fusesoc pgm de0_nano

openocd -f interface/altera-usb-blaster.cfg \
        -f board/or1k_generic.cfg

telnet localhost 4444
> init
> halt; load_image vmlinux ; reset
```

### 4) 在模拟器上运行（可选）

QEMU 是一个处理器仿真器，我们推荐它来模拟 OpenRISC 平台。请按照 QEMU 网站上的 OpenRISC 说明，让 Linux 在 QEMU 上运行。你可以自己构建 QEMU，但你的 Linux 发行版可能提供了支持 OpenRISC 的二进制包。

qemu openrisc	<a href="https://wiki.qemu.org/Documentation/Platforms/OpenRISC">https://wiki.qemu.org/Documentation/Platforms/OpenRISC</a>
---------------	---

## 术语表

代码中使用了以下符号约定以将范围限制在几个特定处理器实现上：

openrisc:	OpenRISC 类型处理器
or1k:	OpenRISC 1000 系列处理器
or1200:	OpenRISC 1200 处理器

## 历史

### 2003-11-18 Matjaz Breskvar ([phoenix@bsemi.com](mailto:phoenix@bsemi.com))

将 linux 初步移植到 OpenRISC 或 32 架构。所有的核心功能都实现了，并且可以使用。

**2003-12-08 Matjaz Breskvar ([phoenix@bsemi.com](mailto:phoenix@bsemi.com))** 彻底改变 TLB 失误处理。重写异常处理。  
在默认的 initrd 中实现了 sash-3.6 的所有功能。大幅改进的版本。

**2004-04-10 Matjaz Breskvar ([phoenix@bsemi.com](mailto:phoenix@bsemi.com))** 大量的 bug 修复。支持以太网，http 和 telnet 服务器功能。可以运行许多标准的 linux 应用程序。

**2004-06-26 Matjaz Breskvar ([phoenix@bsemi.com](mailto:phoenix@bsemi.com))** 移植到 2.6.x。

**2004-11-30 Matjaz Breskvar ([phoenix@bsemi.com](mailto:phoenix@bsemi.com))** 大量的 bug 修复和增强功能。增加了 opencores framebuffer 驱动。

**2010-10-09 Jonas Bonn ([jonas@southpole.se](mailto:jonas@southpole.se))** 重大重写，使其与上游的 Linux 2.6.36 看齐。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/openrisc/todo.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

### 待办事项

OpenRISC Linux 的移植已经完全投入使用，并且从 2.6.35 开始就一直在上游同步。然而，还有一些剩余的项目需要在未来几个月内完成。下面是一个即将进行调查的已知不尽完美的项目列表，即我们的待办事项列表。

- 实现其余的 DMA API……dma\_map\_sg 等。
- 完成重命名清理工作……代码中提到了 or32，这是架构的一个老名字。我们已经确定的名字是 or1k，这个改变正在以缓慢积累的方式进行。目前，or32 相当于 or1k。

### Todolist: features

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/parisc/index.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

### \* PA-RISC 体系架构

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/parisc/debugging.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## 调试 PA-RISC

好吧，这里有一些关于调试 linux/parisc 的较底层部分的信息。

### 1. 绝对地址

很多汇编代码目前运行在实模式下，这意味着会使用绝对地址，而不是像内核其他部分那样使用虚拟地址。要将绝对地址转换为虚拟地址，你可以在 System.map 中查找，添加 \_\_PAGE\_OFFSET（目前是 0x10000000）。

### 2. HPMCs

当实模式的代码试图访问不存在的内存时，会出现 HPMC（high priority machine check）而不是内核 oops。若要调试 HPMC，请尝试找到系统响应程序/请求程序地址。系统请求程序地址应该与（某）处理器的 HPA（I/O 范围内的高地址）相匹配；系统响应程序地址是实模式代码试图访问的地址。

系统响应程序地址的典型值是大于 \_\_PAGE\_OFFSET（0x10000000）的地址，这意味着在实模式试图访问它之前，虚拟地址没有被翻译成物理地址。

### 3. 有趣的 Q 位

某些非常关键的代码必须清除 PSW 中的 Q 位。当 Q 位被清除时，CPU 不会更新中断处理程序所读取的寄存器，以找出机器被中断的位置——所以如果你在清除 Q 位的指令和再次设置 Q 位的 RFI 之间遇到中断，你不知道它到底发生在哪。如果你幸运的话，IAOQ 会指向清除 Q 位的指令，如果你不幸运的话，它会指向任何地方。通常 Q 位的问题会表现为无法解释的系统挂起或物理内存越界。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/parisc/registers.rst

翻译 司延腾 Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)>

## Linux/PA-RISC 的寄存器用法

[ 用星号表示目前尚未实现的计划用途。 ]

### ABI 约定的通用寄存器

#### 控制寄存器

CR 0 (恢复计数器)	用于 ptrace
CR 1-CR 7(无定义)	未使用
CR 8 (Protection ID)	每进程值 *
CR 9, 12, 13 (PIDS)	未使用
CR10 (CCR)	FPU 延迟保存 *
CR11	按照 ABI 的规定 (SAR)
CR14 (中断向量)	初始化为 fault_vector
CR15 (EIEM)	所有位初始化为 1*
CR16 (间隔计时器)	读取周期数/写入开始时间间隔计时器
CR17-CR22	中断参数
CR19	中断指令寄存器
CR20	中断空间寄存器
CR21	中断偏移量寄存器
CR22	中断 PSW
CR23 (EIRR)	读取未决中断/写入清除位
CR24 (TR 0)	内核空间页目录指针
CR25 (TR 1)	用户空间页目录指针
CR26 (TR 2)	不使用
CR27 (TR 3)	线程描述符指针
CR28 (TR 4)	不使用
CR29 (TR 5)	不使用
CR30 (TR 6)	当前 / 0
CR31 (TR 7)	临时寄存器, 在不同地方使用

#### 空间寄存器 (内核模式)

SR0	临时空间寄存器
SR4-SR7	设置为 0
SR1	临时空间寄存器
SR2	内核不应该破坏它
SR3	用于用户空间访问 (当前进程)

## 空间寄存器（用户模式）

SR0	临时空间寄存器
SR1	临时空间寄存器
SR2	保存 Linux gateway page 的空间
SR3	在内核中保存用户地址空间的值
SR4-SR7	定义了用户/内核的短地址空间

## 处理器状态字

W (64 位地址)	0
E (小尾端)	0
S (安全间隔计时器)	0
T (产生分支陷阱)	0
H (高特权级陷阱)	0
L (低特权级陷阱)	0
N (撤销下一条指令)	被 C 代码使用
X (数据存储中断禁用)	0
B (产生分支)	被 C 代码使用
C (代码地址转译)	1, 在执行实模式代码时为 0
V (除法步长校正)	被 C 代码使用
M (HPMC 掩码)	0, 在执行 HPMC 操作 * 时为 1
C/B (进/借位)	被 C 代码使用
O (有序引用)	1*
F (性能监视器)	0
R (回收计数器陷阱)	0
Q (收集中断状态)	1 (在 rfi 之前的代码中为 0)
P (保护标识符)	1*
D (数据地址转译)	1, 在执行实模式代码时为 0
I (外部中断掩码)	由 cli()/sti() 宏使用。

## “隐形” 寄存器（影子寄存器）

PSW W 默认值	0
PSW E 默认值	0
影子寄存器	被中断处理代码使用
TOC 启用位	1

PA-RISC 架构定义了 7 个寄存器作为“影子寄存器”。这些寄存器在 RETURN FROM INTERRUPTION AND RESTORE 指令中使用，通过消除中断处理程序中对一般寄存器（GR）的保存和恢复的需要来减少状态保存和恢复时间。影子寄存器是 GRs 1, 8, 9, 16, 17, 24 和 25。

---

寄存器使用说明，最初由 John Marvin 提供，并由 Randolph Chung 提供一些补充说明。

对于通用寄存器：

r1,r2,r19-r26,r28,r29 & r31 可以在不保存它们的情况下被使用。当然，如果你关心它们，在调用另一个程序之前，你也需要保存它们。上面的一些寄存器确实有特殊的含义，你应该注意一下：

**r1:** addil 指令是硬性规定将其结果放在 r1 中，所以如果你使用这条指令要注意这点。

**r2:** 这就是返回指针。一般来说，你不想使用它，因为你需要这个指针来返回给你的调用者。然而，它与这组寄存器组合在一起，因为调用者不能依赖你返回时的值是相同的，也就是说，你可以将 r2 复制到另一个寄存器，并在作废 r2 后通过该寄存器返回，这应该不会给调用程序带来问题。

**r19-r22:** 这些通常被认为是临时寄存器。请注意，在 64 位中它们是 arg7-arg4。

**r23-r26:** 这些是 arg3-arg0，也就是说，如果你不再关心传入的值，你可以使用它们。

**r28,r29:** 这俩是 ret0 和 ret1。它们是你传入返回值的地方。r28 是主返回值。当返回小结构体时，r29 也可以用来将数据传回给调用程序。

**r30:** 栈指针

**r31:** ble 指令将返回指针放在这里。

r3-r18,r27,r30 需要被保存和恢复。r3-r18 只是一般用途的寄存器。r27 是数据指针，用来使对全局变量的引用更容易。r30 是栈指针。

Todolist:

features

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

---

**Original** Documentation/loongarch/index.rst

**Translator** Huacai Chen <[chenhuacai@loongson.cn](mailto:chenhuacai@loongson.cn)>

## \* LoongArch 体系结构

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/loongarch/introduction.rst

**Translator** Huacai Chen <[chenhuacai@loongson.cn](mailto:chenhuacai@loongson.cn)>

## LoongArch 介绍

LoongArch 是一种新的 RISC ISA，在一定程度上类似于 MIPS 和 RISC-V。LoongArch 指令集包括一个精简 32 位版 (LA32R)、一个标准 32 位版 (LA32S)、一个 64 位版 (LA64)。LoongArch 定义了四个特权级 (PLV0~PLV3)，其中 PLV0 是最高特权级，用于内核；而 PLV3 是最低特权级，用于应用程序。本文档介绍了 LoongArch 的寄存器、基础指令集、虚拟内存以及其他一些主题。

## 寄存器

LoongArch 的寄存器包括通用寄存器 (GPRs)、浮点寄存器 (FPRs)、向量寄存器 (VRs) 和用于特权模式 (PLV0) 的控制状态寄存器 (CSRs)。

### 通用寄存器

LoongArch 包括 32 个通用寄存器 (\$r0 ~ \$r31)，LA32 中每个寄存器为 32 位宽，LA64 中每个寄存器为 64 位宽。\$r0 的内容总是固定为 0，而其他寄存器在体系结构层面没有特殊功能。（\$r1 算是一个例外，在 BL 指令中固定用作链接返回寄存器。）

内核使用了一套 LoongArch 寄存器约定，定义在 LoongArch ELF psABI 规范中，详细描述参见[参考文献](#)：

寄存器名	别名	用途	跨调用保持
\$r0	\$zero	常量 0	不使用
\$r1	\$ra	返回地址	否
\$r2	\$tp	TLS/线程信息指针	不使用
\$r3	\$sp	栈指针	是
\$r4-\$r11	\$a0-\$a7	参数寄存器	否
\$r4-\$r5	\$v0-\$v1	返回值	否
\$r12-\$r20	\$t0-\$t8	临时寄存器	否
\$r21	\$u0	每 CPU 变量基地址	不使用
\$r22	\$fp	帧指针	是
\$r23-\$r31	\$s0-\$s8	静态寄存器	是

**Note:** 注意: \$r21 寄存器在 ELF psABI 中保留未使用，但是在 Linux 内核用于保存每 CPU 变量基地址。该寄存器没有 ABI 命名，不过在内核中称为 \$u0。在一些遗留代码中有时可能见到 \$v0 和 \$v1，它们是 \$a0 和 \$a1 的别名，属于已经废弃的用法。

## 浮点寄存器

当系统中存在 FPU 时，LoongArch 有 32 个浮点寄存器 (\$f0 ~ \$f31)。在 LA64 的 CPU 核上，每个寄存器均为 64 位宽。

浮点寄存器的使用约定与 LoongArch ELF psABI 规范的描述相同：

寄存器名	别名	用途	跨调用保持
\$f0-\$f7	\$fa0-\$fa7	参数寄存器	否
\$f0-\$f1	\$fv0-\$fv1	返回值	否
\$f8-\$f23	\$ft0-\$ft15	临时寄存器	否
\$f24-\$f31	\$fs0-\$fs7	静态寄存器	是

**Note:** 注意: 在一些遗留代码中有时可能见到 \$v0 和 \$v1，它们是 \$a0 和 \$a1 的别名，属于已经废弃的用法。

## 向量寄存器

LoongArch 现有两种向量扩展:

- 128 位向量扩展 LSX (全称 Loongson SIMD eXtention),
- 256 位向量扩展 LASX (全称 Loongson Advanced SIMD eXtention)。

LSX 使用 \$v0 ~ \$v31 向量寄存器, 而 LASX 则使用 \$x0 ~ \$x31。

浮点寄存器和向量寄存器是复用的, 比如: 在一个实现了 LSX 和 LASX 的核上, \$x0 的低 128 位与 \$v0 共用, \$v0 的低 64 位与 \$f0 共用, 其他寄存器依此类推。

## 控制状态寄存器

控制状态寄存器只能在特权模式 (PLV0) 下访问:

地址	全称描述	简称
0x0	当前模式信息	CRMD
0x1	异常前模式信息	PRMD
0x2	扩展部件使能	EUEN
0x3	杂项控制	MISC
0x4	异常配置	ECFG
0x5	异常状态	ESTAT
0x6	异常返回地址	ERA
0x7	出错 (Faulting) 虚拟地址	BADV
0x8	出错 (Faulting) 指令字	BADI
0xC	异常入口地址	EENTRY
0x10	TLB 索引	TLBIDX
0x11	TLB 表项高位	TLBEHI
0x12	TLB 表项低位 0	TLBELO0
0x13	TLB 表项低位 1	TLBELO1
0x18	地址空间标识符	ASID
0x19	低半地址空间页全局目录基址	PGDL
0x1A	高半地址空间页全局目录基址	PGDH
0x1B	页全局目录基址	PGD
0x1C	页表遍历控制低半部分	PWCL
0x1D	页表遍历控制高半部分	PWCH
0x1E	STLB 页大小	STLBPS
0x1F	缩减虚地址配置	RVACFG
0x20	CPU 编号	CPUID
0x21	特权资源配置信息 1	PRCFG1

continues on next page

Table 5 – continued from previous page

地址	全称描述	简称
0x22	特权资源配置信息 2	PRCFG2
0x23	特权资源配置信息 3	PRCFG3
0x30+n (0≤n≤15)	数据保存寄存器	SAVEn
0x40	定时器编号	TID
0x41	定时器配置	TCFG
0x42	定时器值	TVAL
0x43	计时器补偿	CNTC
0x44	定时器中断清除	TICLR
0x60	LLBit 相关控制	LLBCTL
0x80	实现相关控制 1	IMPCTL1
0x81	实现相关控制 2	IMPCTL2
0x88	TLB 重填异常入口地址	TLBREENTRY
0x89	TLB 重填异常出错 (Faulting) 虚地址	TLBRBADV
0x8A	TLB 重填异常返回地址	TLBRERA
0x8B	TLB 重填异常数据保存	TLBRSAVE
0x8C	TLB 重填异常表项低位 0	TLBRELO0
0x8D	TLB 重填异常表项低位 1	TLBRELO1
0x8E	TLB 重填异常表项高位	TLBEHI
0x8F	TLB 重填异常前模式信息	TLBRPRMD
0x90	机器错误控制	MERRCTL
0x91	机器错误信息 1	MERRINFO1
0x92	机器错误信息 2	MERRINFO2
0x93	机器错误异常入口地址	MERREENTRY
0x94	机器错误异常返回地址	MERRERA
0x95	机器错误异常数据保存	MERRSAVE
0x98	高速缓存标签	CTAG
0x180+n (0≤n≤3)	直接映射配置窗口 n	DMWn
0x200+2n (0≤n≤31)	性能监测配置 n	PMCFGn
0x201+2n (0≤n≤31)	性能监测计数器 n	PMCNTn
0x300	内存读写监视点整体控制	MWPC
0x301	内存读写监视点整体状态	MWPS
0x310+8n (0≤n≤7)	内存读写监视点 n 配置 1	MWPnCFG1
0x311+8n (0≤n≤7)	内存读写监视点 n 配置 2	MWPnCFG2
0x312+8n (0≤n≤7)	内存读写监视点 n 配置 3	MWPnCFG3
0x313+8n (0≤n≤7)	内存读写监视点 n 配置 4	MWPnCFG4
0x380	取指监视点整体控制	FWPC
0x381	取指监视点整体状态	FWPS
0x390+8n (0≤n≤7)	取指监视点 n 配置 1	FWPnCFG1

continues on next page

Table 5 – continued from previous page

地址	全称描述	简称
0x391+8n (0≤n≤7)	取指监视点 n 配置 2	FWPnCFG2
0x392+8n (0≤n≤7)	取指监视点 n 配置 3	FWPnCFG3
0x393+8n (0≤n≤7)	取指监视点 n 配置 4	FWPnCFG4
0x500	调试寄存器	DBG
0x501	调试异常返回地址	DERA
0x502	调试数据保存	DSAVE

ERA, TLBRERA, MERRERA 和 DERA 有时也分别称为 EPC, TLBREPC, MERREPC 和 DEPC。

## 基础指令集

### 指令格式

LoongArch 的指令字长为 32 位，一共有 9 种基本指令格式（以及一些变体）：

格式名称	指令构成
2R	Opcode + Rj + Rd
3R	Opcode + Rk + Rj + Rd
4R	Opcode + Ra + Rk + Rj + Rd
2RI8	Opcode + I8 + Rj + Rd
2RI12	Opcode + I12 + Rj + Rd
2RI14	Opcode + I14 + Rj + Rd
2RI16	Opcode + I16 + Rj + Rd
1RI21	Opcode + I21L + Rj + I21H
I26	Opcode + I26L + I26H

Opcode 是指令操作码，Rj 和 Rk 是源操作数（寄存器），Rd 是目标操作数（寄存器），Ra 是 4R-type 格式特有的附加操作数（寄存器）。I8/I12/I16/I21/I26 分别是 8 位/12 位/16 位/21 位/26 位的立即数。其中较长的 21 位和 26 位立即数在指令字中被分割为高位部分与低位部分，所以你们在这里的格式描述中能够看到 I21L/I21H 和 I26L/I26H 这样带后缀的表述。

## 指令列表

为了简便起见，我们在此只罗列一下指令名称（助记符），需要详细信息请阅读[参考文献](#)中的文档。

### 1. 算术运算指令：

```
ADD.W SUB.W ADDI.W ADD.D SUB.D ADDI.D
SLT SLTU SLTI SLTUI
AND OR NOR XOR ANDN ORN ANDI ORI XORI
```

```
MUL.W MULH.W MULH.WU DIV.W DIV.WU MOD.W MOD.WU  
MUL.D MULH.D MULH.DU DIV.D DIV.DU MOD.D MOD.DU  
PCADDI PCADDU12I PCADDU18I  
LU12I.W LU32I.D LU52I.D ADDU16I.D
```

### 2. 移位运算指令:

```
SLL.W SRL.W SRA.W ROTR.W SLLI.W SRLI.W SRAI.W ROTRI.W  
SLL.D SRL.D SRA.D ROTR.D SLLI.D SRLI.D SRAI.D ROTRI.D
```

### 3. 位域操作指令:

```
EXT.W.B EXT.W.H CLO.W CLO.D SLZ.W CLZ.D CTO.W CTO.D CTZ.W CTZ.D  
BYTEPICK.W BYTEPICK.D BSTRINS.W BSTRINS.D BSTRPICK.W BSTRPICK.D  
REVB.2H REVB.4H REVB.2W REVB.D REVH.2W REVH.D BITREV.4B BITREV.8B BITREV.W BITREV.D  
MASKEQZ MASKNEZ
```

### 4. 分支转移指令:

```
BEQ BNE BLT BGE BLTU BGEU BEQZ BNEZ B BL JIRL
```

### 5. 访存读写指令:

```
LD.B LD.BU LD.H LD.HU LD.W LD.WU LD.D ST.B ST.H ST.W ST.D  
LDX.B LDX.BU LDX.H LDX.HU LDX.W LDX.WU LDX.D STX.B STX.H STX.W STX.D  
LDPTR.W LD PTR.D STPTR.W STPTR.D  
PRELD PRELDX
```

### 6. 原子操作指令:

```
LL.W SC.W LL.D SC.D  
AMSWAP.W AMSWAP.D AMADD.W AMADD.D AMAND.W AMAND.D AMOR.W AMOR.D AMXOR.W AMXOR.D  
AMMAX.W AMMAX.D AMMIN.W AMMIN.D
```

### 7. 栅障指令:

```
IBAR DBAR
```

### 8. 特殊指令:

```
SYSCALL BREAK CPUCFG NOP IDLE ERTN(ERET) DBCL(DBGCALL) RDTIMEL.W RDTIMEH.W RDTIME.D  
ASRTLE.D ASRTGT.D
```

### 9. 特权指令:

```
CSRRD CSRWR CSRXCHG  
IOCSRRD.B IOCSRRD.H IOCSRRD.W IOCSRRD.D IOCSRWR.B IOCSRWR.H IOCSRWR.W IOCSRWR.D  
CACOP TLBP(TLBSRCH) TLBRD TLBWR TLBFILL TLBCLR TLBFLUSH INVTLB LDDIR LDPT
```

## 虚拟内存

LoongArch 可以使用直接映射虚拟内存和分页映射虚拟内存。

直接映射虚拟内存通过 CSR.DMWn ( $n=0\sim3$ ) 来进行配置，虚拟地址 (VA) 和物理地址 (PA) 之间有简单的映射关系：

$$\text{VA} = \text{PA} + \text{固定偏移}$$

分页映射的虚拟地址 (VA) 和物理地址 (PA) 有任意的映射关系，这种关系记录在 TLB 和页表中。LoongArch 的 TLB 包括一个全相联的 MTLB (Multiple Page Size TLB, 多样页大小 TLB) 和一个组相联的 STLB (Single Page Size TLB, 单一页大小 TLB)。

缺省状态下，LA32 的整个虚拟地址空间配置如下：

区段名	地址范围	属性
UVRANGE	0x00000000 - 0x7FFFFFFF	分页映射, 可缓存, PLV0~3
KPRANGE0	0x80000000 - 0x9FFFFFFF	直接映射, 非缓存, PLV0
KPRANGE1	0xA0000000 - 0xBFFFFFFF	直接映射, 可缓存, PLV0
KVRANGE	0xC0000000 - 0xFFFFFFFF	分页映射, 可缓存, PLV0

用户态 (PLV3) 只能访问 UVRANGE，对于直接映射的 KPRANGE0 和 KPRANGE1，将虚拟地址的第 30~31 位清零就等于物理地址。例如：物理地址 0x00001000 对应的非缓存直接映射虚拟地址是 0x80001000，而其可缓存直接映射虚拟地址是 0xA0001000。

缺省状态下，LA64 的整个虚拟地址空间配置如下：

区段名	地址范围	属性
XUVRANGE	0x0000000000000000 - 0x3FFFFFFFFFFFFF	分页映射, 可缓存, PLV0~3
XSPRANGE	0x4000000000000000 - 0x7FFFFFFFFFFFFF	直接映射, 可缓存 / 非缓存, PLV0
XKPRANGE	0x8000000000000000 - 0xBFFFFFFFFFFFFF	直接映射, 可缓存 / 非缓存, PLV0
XKVRANGE	0xC000000000000000 - 0xFFFFFFFFFFFF	分页映射, 可缓存, PLV0

用户态 (PLV3) 只能访问 XUVRANGE，对于直接映射的 XSPRANGE 和 XKPRANGE，将虚拟地址的第 60~63 位清零就等于物理地址，而其缓存属性是通过虚拟地址的第 60~61 位配置的 (0 表示强序非缓存，1 表示一致可缓存，2 表示弱序非缓存)。

目前，我们仅用 XKPRANGE 来进行直接映射，XSPRANGE 保留给以后用。

此处给出一个直接映射的例子：物理地址 0x00000000\_00001000 的强序非缓存直接映射虚拟地址（在 XKPRANGE 中）是 0x80000000\_00001000，其一致可缓存直接映射虚拟地址（在 XKPRANGE 中）是 0x90000000\_00001000，而其弱序非缓存直接映射虚拟地址（在 XKPRANGE 中）是 0xA0000000\_00001000。

### Loongson 与 LoongArch 的关系

LoongArch 是一种 RISC 指令集架构 (ISA)，不同于现存的任何一种 ISA，而 Loongson (即龙芯) 是一个处理器家族。龙芯包括三个系列：Loongson-1 (龙芯 1 号) 是 32 位处理器系列，Loongson-2 (龙芯 2 号) 是低端 64 位处理器系列，而 Loongson-3 (龙芯 3 号) 是高端 64 位处理器系列。旧的龙芯处理器基于 MIPS 架构，而新的龙芯处理器基于 LoongArch 架构。以龙芯 3 号为例：龙芯 3A1000/3B1500/3A2000/3A3000/3A4000 都是兼容 MIPS 的，而龙芯 3A5000 (以及将来的型号) 都是基于 LoongArch 的。

### 参考文献

Loongson 官方网站（龙芯中科技术股份有限公司）：

<http://www.loongson.cn/>

Loongson 与 LoongArch 的开发者网站（软件与文档资源）：

<http://www.loongnix.cn/>

<https://github.com/loongson/>

<https://loongson.github.io/LoongArch-Documentation/>

LoongArch 指令集架构的文档：

<https://github.com/loongson/LoongArch-Documentation/releases/latest/download/LoongArch-Vol1-v1.00-CN.pdf> (中文版)

<https://github.com/loongson/LoongArch-Documentation/releases/latest/download/LoongArch-Vol1-v1.00-EN.pdf> (英文版)

LoongArch 的 ELF psABI 文档：

<https://github.com/loongson/LoongArch-Documentation/releases/latest/download/LoongArch-ELF-ABI-v1.00-CN.pdf> (中文版)

<https://github.com/loongson/LoongArch-Documentation/releases/latest/download/LoongArch-ELF-ABI-v1.00-EN.pdf> (英文版)

Loongson 与 LoongArch 的 Linux 内核源码仓库：

<https://git.kernel.org/pub/scm/linux/kernel/git/chenhuacai/linux-loongson.git>

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

---

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的

帮助: <[alexs@kernel.org](mailto:alexs@kernel.org)>。

**Original** Documentation/loongarch/irq-chip-model.rst

**Translator** Huacai Chen <[chenhuacai@loongson.cn](mailto:chenhuacai@loongson.cn)>

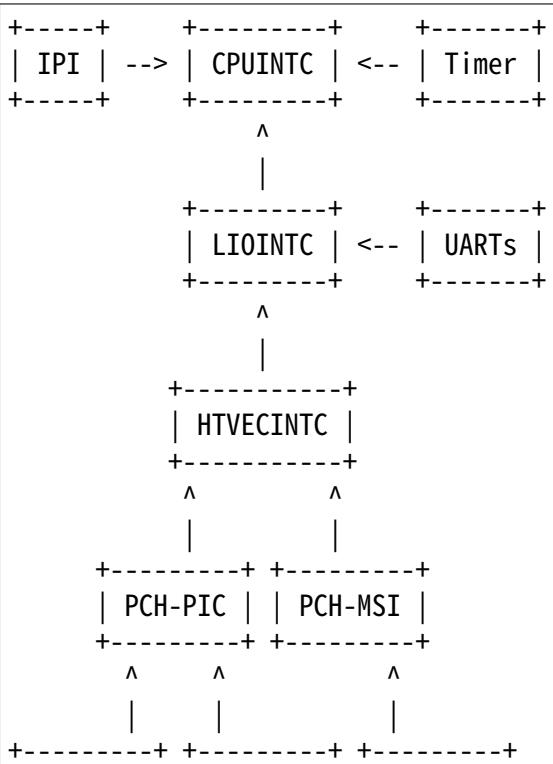
## LoongArch 的 IRQ 芯片模型（层级关系）

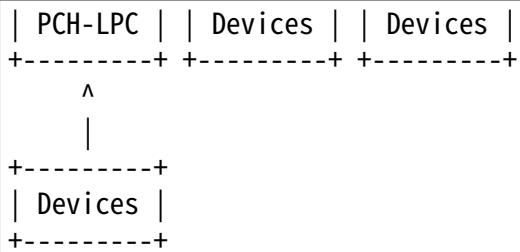
目前，基于 LoongArch 的处理器（如龙芯 3A5000）只能与 LS7A 芯片组配合工作。LoongArch 计算机中的中断控制器（即 IRQ 芯片）包括 CPUINTC (CPU Core Interrupt Controller)、LIOINTC (Legacy I/O Interrupt Controller)、EIOINTC (Extended I/O Interrupt Controller)、HTVECINTC (Hyper-Transport Vector Interrupt Controller)、PCH-PIC (LS7A 芯片组的主中断控制器)、PCH-LPC (LS7A 芯片组的 LPC 中断控制器) 和 PCH-MSI (MSI 中断控制器)。

CPUINTC 是一种 CPU 内部的每个核本地的中断控制器，LIOINTC/EIOINTC/HTVECINTC 是 CPU 内部的全局中断控制器（每个芯片一个，所有核共享），而 PCH-PIC/PCH-LPC/PCH-MSI 是 CPU 外部的中断控制器（在配套芯片组里面）。这些中断控制器（或者说 IRQ 芯片）以一种层次树的组织形式级联在一起，一共有两种层级关系模型（传统 IRQ 模型和扩展 IRQ 模型）。

### 传统 IRQ 模型

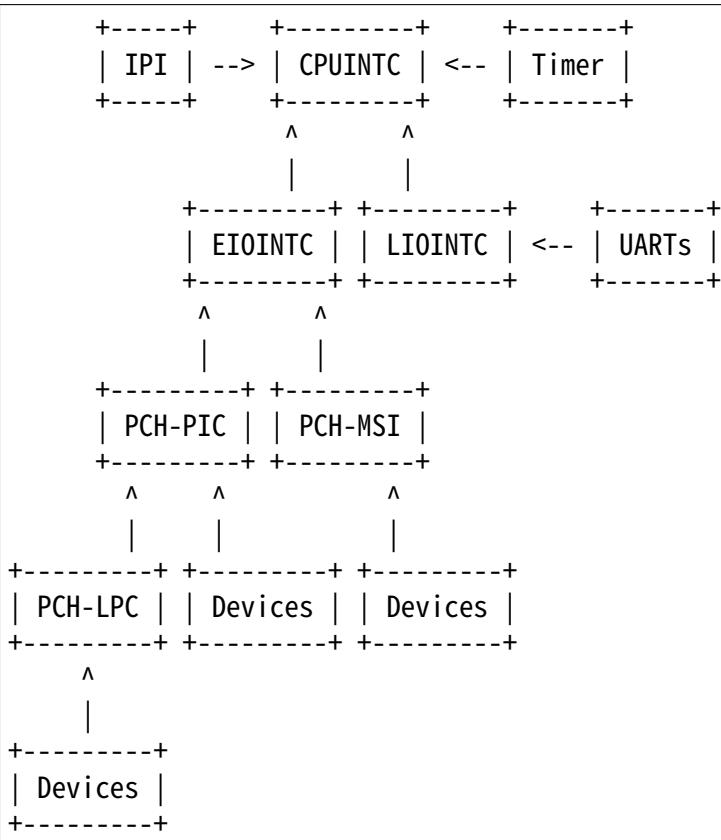
在这种模型里面，IPI (Inter-Processor Interrupt) 和 CPU 本地时钟中断直接发送到 CPUINTC，CPU 串口 (UARTs) 中断发送到 LIOINTC，而其他所有设备的中断则分别发送到所连接的 PCH-PIC/ PCH-LPC/PCH-MSI，然后被 HTVECINTC 统一收集，再发送到 LIOINTC，最后到达 CPUINTC：





## 扩展 IRQ 模型

在这种模型里面，IPI（Inter-Processor Interrupt）和 CPU 本地时钟中断直接发送到 CPUINTC，CPU 串口（UARTs）中断发送到 LIOINTC，而其他所有设备的中断则分别发送到所连接的 PCH-PIC/ PCH-LPC/PCH-MSI，然后被 EIOINTC 统一收集，再直接到达 CPUINTC：



## ACPI 相关的定义

CPUINTC:

```

ACPI_MADT_TYPE_CORE_PIC;
struct acpi_madt_core_pic;
enum acpi_madt_core_pic_version;
  
```

LIOINTC:

```
ACPI_MADT_TYPE_LIO_PIC;
struct acpi_madt_lio_pic;
enum acpi_madt_lio_pic_version;
```

EIOINTC:

```
ACPI_MADT_TYPE_EIO_PIC;
struct acpi_madt_eio_pic;
enum acpi_madt_eio_pic_version;
```

HTVECINTC:

```
ACPI_MADT_TYPE_HT_PIC;
struct acpi_madt_ht_pic;
enum acpi_madt_ht_pic_version;
```

PCH-PIC:

```
ACPI_MADT_TYPE_BIO_PIC;
struct acpi_madt_bio_pic;
enum acpi_madt_bio_pic_version;
```

PCH-MSI:

```
ACPI_MADT_TYPE_MSI_PIC;
struct acpi_madt_msi_pic;
enum acpi_madt_msi_pic_version;
```

PCH-LPC:

```
ACPI_MADT_TYPE_LPC_PIC;
struct acpi_madt_lpc_pic;
enum acpi_madt_lpc_pic_version;
```

## 参考文献

龙芯 3A5000 的文档:

<https://github.com/loongson/LoongArch-Documentation/releases/latest/download/Loongson-3A5000-usermanual-1.02-CN.pdf> (中文版)

<https://github.com/loongson/LoongArch-Documentation/releases/latest/download/Loongson-3A5000-usermanual-1.02-EN.pdf> (英文版)

龙芯 LS7A 芯片组的文档:

<https://github.com/loongson/LoongArch-Documentation/releases/latest/download/Loongson-7A1000-usermanual-2.00-CN.pdf> (中文版)

<https://github.com/loongson/LoongArch-Documentation/releases/latest/download/Loongson-7A1000-usermanual-2.00-EN.pdf> (英文版)

### Note:

- CPUINTC: 即《龙芯架构参考手册卷一》第 7.4 节所描述的 CSR.ECFG/CSR.ESTAT 寄存器及其中断控制逻辑；
- LIOINTC: 即《龙芯 3A5000 处理器使用手册》第 11.1 节所描述的“传统 I/O 中断”；
- EIOINTC: 即《龙芯 3A5000 处理器使用手册》第 11.2 节所描述的“扩展 I/O 中断”；
- HTVECINTC: 即《龙芯 3A5000 处理器使用手册》第 14.3 节所描述的“HyperTransport 中断”；
- PCH-PIC/PCH-MSI: 即《龙芯 7A1000 桥片用户手册》第 5 章所描述的“中断控制器”；
- PCH-LPC: 即《龙芯 7A1000 桥片用户手册》第 24.3 节所描述的“LPC 中断”。

**Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

**Note:** 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：[<alexs@kernel.org>](mailto:alexs@kernel.org)。

**Original** Documentation/loongarch/features.rst

**Translator** Huacai Chen <[chenhuacai@loongson.cn](mailto:chenhuacai@loongson.cn)>

### Feature status on loongarch architecture

Subsystem	Feature	Kconfig	Status	Description
core	cBPF-JIT	HAVE_CBPF_JIT		arch
core	eBPF-JIT	HAVE_EBPF_JIT		arch
core	generic-idle-thread	GENERIC_SMP_IDLE_THREAD		arch
core	jump-labels	HAVE_ARCH_JUMP_LABEL		arch
core	thread-info-in-task	THREAD_INFO_IN_TASK		arch
core	tracehook	HAVE_ARCH_TRACEHOOK		arch
debug	debug-vm-pgtable	ARCH_HAS_DEBUG_VM_PGTABLE		arch
debug	gcov-profile-all	ARCH_HAS_GCOV_PROFILE_ALL		arch
debug	KASAN	HAVE_ARCH_KASAN		arch

Table 6 – continued from p

Subsystem	Feature	Kconfig	Status	Description
debug	kcov	ARCH_HAS_KCOV		arch
debug	kgdb	HAVE_ARCH_KGDB		arch
debug	kmemleak	HAVE_DEBUG_KMEMLEAK		arch
debug	kprobes	HAVE_KPROBES		arch
debug	kprobes-on-ftrace	HAVE_KPROBES_ON_FTRACE		arch
debug	kretprobes	HAVE_KRETPROBES		arch
debug	optprobes	HAVE_OPTPROBES		arch
debug	stackprotector	HAVE_STACKPROTECTOR		arch
debug	uprobes	ARCH_SUPPORTS_UPROBES		arch
debug	user-ret-profiler	HAVE_USER_RETURN_NOTIFIER		arch
io	dma-contiguous	HAVE_DMA_CONTIGUOUS		arch
locking	cmpxchg-local	HAVE_CMPXCHG_LOCAL		arch
locking	lockdep	LOCKDEP_SUPPORT		arch
locking	queued-rwlocks	ARCH_USE_QUEUED_RWLOCKS		arch
locking	queued-spinlocks	ARCH_USE_QUEUED_SPINLOCKS		arch
perf	kprobes-event	HAVE_REGS_AND_STACK_ACCESS_API		arch
perf	perf-reg	HAVE_PERF_REGS		arch
perf	perf-stackdump	HAVE_PERF_USER_STACK_DUMP		arch
sched	membarrier-sync-core	ARCH_HAS_MEMARRIER_SYNC_CORE		arch
sched	numa-balancing	ARCH_SUPPORTS_NUMA_BALANCING		arch
seccomp	seccomp-filter	HAVE_ARCH_SECCOMP_FILTER		arch
time	arch-tick-broadcast	ARCH_HAS_TICK_BROADCAST		arch
time	clockevents	!LEGACY_TIMER_TICK		arch
time	context-tracking	HAVE_CONTEXT_TRACKING		arch
time	irq-time-acct	HAVE_IRQ_TIME_ACCOUNTING		arch
time	virt-cpuacct	HAVE_VIRT_CPU_ACCOUNTING		arch
vm	batch-unmap-tlb-flush	ARCH_WANT_BATCHED_UNMAP_TLB_FLUSH		arch
vm	ELF-ASLR	ARCH_HAS_ELF_RANDOMIZE		arch
vm	huge-vmap	HAVE_ARCH_HUGE_VMAP		arch
vm	ioremap_prot	HAVE_IOREMAP_PROT		arch
vm	PG_uncached	ARCHUSES_PG_UNCACHED		arch
vm	pte_special	ARCH_HAS_PTE_SPECIAL		arch
vm	THP	HAVE_ARCH_TRANSPARENT_HUGEPAGE		arch

TODOList:

- arm/index
- ia64/index
- m68k/index

- nios2/index
- powerpc/index
- s390/index
- sh/index
- sparc/index
- x86/index
- xtensa/index

## \* 其他文档

有几份未排序的文档似乎不适合放在文档的其他部分，或者可能需要进行一些调整和/或转换为 reStructureText 格式，也有可能太旧。

TODOList:

- staging/index
- watch\_queue

## \* 目录和表格

- genindex

## 繁體中文翻譯

---

**Note:** 內核文檔繁體中文版的翻譯工作正在進行中。如果您願意並且有時間參與這項工作，歡迎提交補丁給胡皓文 <src.res@email.cn>。

---

### \* 許可證文檔

下面的文檔介紹了 Linux 內核原始碼的許可證 (GPLv2)、如何在原始碼樹中正確標記單個文件的許可證、以及指向完整許可證文本的連結。

*Linux 內核許可規則*

### \* 用戶文檔

下面的手冊是為內核用戶編寫的——即那些試圖讓它在給定系統上以最佳方式工作的用戶。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

---

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：<src.res@email.cn>。

---

**Original** ../../admin-guide/index

**Translator** 胡皓文 Hu Haowen <src.res@email.cn>

### \* Linux 內核用戶和管理員指南

下面是一組隨時間添加到內核中的面向用戶的文檔的集合。到目前為止，還沒有一個整體的順序或組織 - 這些材料不是一個單一的，連貫的文件！幸運的話，情況會隨著時間的推移而迅速改善。

這個初始部分包含總體信息，包括描述內核的 README，關於內核參數的文檔等。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

---

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

---

**Original** Documentation/admin-guide/README.rst

譯者 吳 想 成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)> 胡 皓 文 Hu Haowen  
<[src.res@email.cn](mailto:<src.res@email.cn>)>

### Linux 內核 5.x 版本 <<http://kernel.org/>>

以下是 Linux 版本 5 的發行註記。仔細閱讀它們，它們會告訴你這些都是什麼，解釋如何安裝內核，以及遇到問題時該如何做。

### 什麼是 Linux？

Linux 是 Unix 作業系統的克隆版本，由 Linus Torvalds 在一個鬆散的網絡黑客（Hacker，無貶義）團隊的幫助下從頭開始編寫。它旨在實現兼容 POSIX 和單一 UNIX 規範。

它具有在現代成熟的 Unix 中應當具有的所有功能，包括真正的多任務處理、虛擬內存、共享庫、按需加載、共享的寫時拷貝（COW）可執行文件、恰當的內存管理以及包括 IPv4 和 IPv6 在內的複合網絡棧。

Linux 在 GNU 通用公共許可證，版本 2 (GNU GPLv2) 下分發，詳見隨附的 COPYING 文件。

## 它能在什麼樣的硬體上運行？

雖然 Linux 最初是為 32 位的 x86 PC 機（386 或更高版本）開發的，但今天它也能運行在（至少）Compaq Alpha AXP、Sun SPARC 與 UltraSPARC、Motorola 68000、PowerPC、PowerPC64、ARM、Hitachi SuperH、Cell、IBM S/390、MIPS、HP PA-RISC、Intel IA-64、DEC VAX、AMD x86-64 Xtensa 和 ARC 架構上。

Linux 很容易移植到大多數通用的 32 位或 64 位體系架構，只要它們有一個分頁內存管理單元（PMMU）和一個移植的 GNU C 編譯器（gcc；GNU Compiler Collection，GCC 的一部分）。Linux 也被移植到許多沒有 PMMU 的體系架構中，儘管功能顯然受到了一定的限制。Linux 也被移植到了其自己上。現在可以將內核作為用戶空間應用程式運行——這被稱為用戶模式 Linux（UML）。

## 文檔

網際網路上和書籍上都有大量的電子文檔，既有 Linux 專屬文檔，也有與一般 UNIX 問題相關的文檔。我建議在任何 Linux FTP 站點上查找 LDP（Linux 文檔項目）書籍的文檔子目錄。本自述文件並不是關於系統的文檔：有更好的可用資源。

- 網際網路上和書籍上都有大量的（電子）文檔，既有 Linux 專屬文檔，也有與普通 UNIX 問題相關的文檔。我建議在任何有 LDP（Linux 文檔項目）書籍的 Linux FTP 站點上查找文檔子目錄。本自述文件並不是關於系統的文檔：有更好的可用資源。
- 文檔/子目錄中有各種自述文件：例如，這些文件通常包含一些特定驅動程序的內核安裝說明。請閱讀 Documentation/process/changes.rst 文件，它包含了升級內核可能會導致的問題的相關信息。

## 安裝內核原始碼

- 如果您要安裝完整的原始碼，請把內核 tar 檔案包放在您有權限的目錄中（例如您的主目錄）並將其解包：

```
xz -cd linux-5.x.tar.xz | tar xvf -
```

將「X」替換成最新內核的版本號。

**【不要】**使用 /usr/src/linux 目錄！這裡有一組庫頭文件使用的內核頭文件（通常是不完整的）。它們應該與庫匹配，而不是被內核的變化搞得一團糟。

- 您還可以通過打補丁在 5.x 版本之間升級。補丁以 xz 格式分發。要通過打補丁進行安裝，請獲取所有較新的補丁文件，進入內核原始碼（linux-5.x）的目錄並執行：

```
xz -cd ../patch-5.x.xz | patch -p1
```

請**【按順序】**替換所有大於當前原始碼樹版本的「X」，這樣就可以了。您可能想要刪除備份文件（文件名類似 xxx~ 或 xxx.orig），並確保沒有失敗的補丁（文件名類似 xxx# 或 xxx.rej）。如果有，不是你就是我犯了錯誤。

與 5.x 內核的補丁不同，5.x.y 內核（也稱為穩定版內核）的補丁不是增量的，而是直接應用於基本的 5.x 內核。例如，如果您的基本內核是 5.0，並且希望應用 5.0.3 補丁，則不應先應用 5.0.1 和 5.0.2 的補丁。類似地，如果您運行的是 5.0.2 內核，並且希望跳轉到 5.0.3，那麼在應用 5.0.3 補丁之前，必須首先撤銷 5.0.2 補丁（即 `patch -R`）。更多關於這方面的內容，請閱讀 `Documentation/process/applying-patches.rst`。

或者，腳本 `patch-kernel` 可以用來自動化這個過程。它能確定當前內核版本並應用找到的所有補丁：

```
linux/scripts/patch-kernel linux
```

上面命令中的第一個參數是內核原始碼的位置。補丁是在當前目錄應用的，但是可以將另一個目錄指定為第二個參數。

- 確保沒有過時的.o 文件和依賴項：

```
cd linux  
make mrproper
```

現在您應該已經正確安裝了原始碼。

## 軟體要求

編譯和運行 5.x 內核需要各種軟體包的最新版本。請參考 `Documentation/process/changes.rst` 來了解最低版本要求以及如何升級軟體包。請注意，使用過舊版本的這些包可能會導致很難追蹤的間接錯誤，因此不要以為在生成或操作過程中出現明顯問題時可以只更新包。

## 為內核建立目錄

編譯內核時，默認情況下所有輸出文件都將與內核原始碼放在一起。使用 `make O=output/dir` 選項可以為輸出文件（包括`.config`）指定備用位置。例如：

```
kernel source code: /usr/src/linux-5.x  
build directory: /home/name/build/kernel
```

要配置和構建內核，請使用：

```
cd /usr/src/linux-5.x  
make O=/home/name/build/kernel menuconfig  
make O=/home/name/build/kernel  
sudo make O=/home/name/build/kernel modules_install install
```

請注意：如果使用了 `O=output/dir` 選項，那麼它必須用於 `make` 的所有調用。

## 配置內核

即使只升級一個小版本，也不要跳過此步驟。每個版本中都會添加新的配置選項，如果配置文件沒有按預定設置，就會出現奇怪的問題。如果您想以最少的工作量將現有配置升級到新版本，請使用 `makeoldconfig`，它只會詢問您新配置選項的答案。

- 其他配置命令包括：

<code>"make config"</code>	純文本界面。
<code>"make menuconfig"</code>	基於文本的彩色菜單、選項列表和對話框。
<code>"make nconfig"</code>	增強的基於文本的彩色菜單。
<code>"make xconfig"</code>	基於 Qt 的配置工具。
<code>"make gconfig"</code>	基於 GTK+ 的配置工具。
<code>"make oldconfig"</code>	基於現有的 <code>./.config</code> 文件選擇所有選項，並詢問新配置選項。
<code>"make olddefconfig"</code>	類似上一個，但不詢問直接將新選項設置為默認值。
<code>"make defconfig"</code>	根據體系架構，使用 <code>arch/\$arch/defconfig</code> 或 <code>arch/\$arch/configs/\${PLATFORM}_defconfig</code> 中的默認選項值創建 <code>./.config</code> 文件。
<code>"make \${PLATFORM}_defconfig"</code>	使用 <code>arch/\$arch/configs/\${PLATFORM}_defconfig</code> 中的默認選項值創建一個 <code>./.config</code> 文件。 用「 <code>makehelp</code> 」來獲取您體系架構中所有可用平台的列表。
<code>"make allyesconfig"</code>	通過儘可能將選項值設置為「y」，創建一個 <code>./.config</code> 文件。
<code>"make allmodconfig"</code>	通過儘可能將選項值設置為「m」，創建一個 <code>./.config</code> 文件。
<code>"make allnoconfig"</code>	通過儘可能將選項值設置為「n」，創建一個 <code>./.config</code> 文件。
<code>"make randconfig"</code>	通過隨機設置選項值來創建 <code>./.config</code> 文件。
<code>"make localmodconfig"</code>	基於當前配置和加載的模塊 ( <code>lsmod</code> ) 創建配置。禁用已加載的模塊不需要的任何模塊選項。
要為另一台計算機創建 <code>localmodconfig</code> ，請將該計算機	

的 `lsmod` 存儲到一個文件中，並將其作為 `lsmod` 參數傳入。

此外，通過在參數 `LMC_KEEP` 中指定模塊的路徑，可以將模塊保留在某些文件夾或 `kconfig` 文件中。

```
target$ lsmod > /tmp/my_lsmod
target$ scp /tmp/my_lsmod host:/tmp

host$ make LSMOD=/tmp/my_lsmod \
      LMC_KEEP="drivers/usb:drivers/gpu:fs" \
      localmodconfig
```

上述方法在交叉編譯時也適用。

`"make localyesconfig"` 與 `localmodconfig` 類似，只是它會將所有模塊選項轉換為內置 (`=y`)。你可以同時通過 `LMC_KEEP` 保留模塊。

`"make kvmconfig"` 為 `kvm` 客體內核支持啓用其他選項。

`"make xenconfig"` 為 `xen dom0` 客體內核支持啓用其他選項。

`"make tinyconfig"` 配置儘可能小的內核。

更多關於使用 Linux 內核配置工具的信息，見文檔 `Documentation/kbuild/kconfig.rst`。

- `make config` 注意事項:

- 包含不必要的驅動程序會使內核變大，並且在某些情況下會導致問題：探測不存在的控制器卡可能會混淆其他控制器。
- 如果存在協處理器，則編譯了數學仿真的內核仍將使用協處理器：在這種情況下，數學仿真永遠不會被使用。內核會稍微大一點，但不管是否有數學協處理器，都可以在不同的機器上工作。
- 「kernel hacking」配置細節通常會導致更大或更慢的內核（或兩者兼而有之），甚至可以通過配置一些例程來主動嘗試破壞代碼以發現內核問題，從而降低內核的穩定性 (`kmalloc()`)。因此，您可能應該用於研究「開發」、「實驗」或「調試」特性相關問題。

## 編譯內核

- 確保您至少有 `gcc 5.1` 可用。有關更多信息，請參閱 `Documentation/process/changes.rst`。  
請注意，您仍然可以使用此內核運行 `a.out` 用戶程序。
- 執行 `make` 來創建壓縮內核映像。如果您安裝了 `lilo` 以適配內核 `makefile`，那麼也可以進行 `makeinstall`，但是您可能需要先檢查特定的 `lilo` 設置。  
實際安裝必須以 `root` 身份執行，但任何正常構建都不需要。無須徒然使用 `root` 身份。
- 如果您將內核的任何部分配置為模塊，那麼還必須執行 `make modules_install`。

- 詳細的內核編譯/生成輸出：

通常，內核構建系統在相當安靜的模式下運行（但不是完全安靜）。但是有時您或其他內核開發人員需要看到編譯、連結或其他命令的執行過程。為此，可使用「verbose（詳細）」構建模式。向 `make` 命令傳遞 `V=1` 來實現，例如：

```
make V=1 all
```

如需構建系統也給出內個目標重建的願意，請使用 `V=2`。默認為 `V=0`。

- 準備一個備份內核以防出錯。對於開發版本尤其如此，因為每個新版本都包含尚未調試的新代碼。也要確保保留與該內核對應的模塊的備份。如果要安裝與工作內核版本號相同的新內核，請在進行 `make modules_install` 安裝之前備份 `modules` 目錄。

或者，在編譯之前，使用內核配置選項「LOCALVERSION」向常規內核版本附加一個唯一的後綴。`LOCALVERSION` 可以在「General Setup」菜單中設置。

- 為了引導新內核，您需要將內核映像（例如編譯後的`../linux/arch/x86/boot/bzImage`）複製到常規可引導內核的位置。
- 不再支持在沒有 LILO 等啟動裝載程序幫助的情況下直接從軟盤引導內核。

如果從硬碟引導 Linux，很可能使用 LILO，它使用`/etc/lilo.conf` 文件中指定的內核映像文件。內核映像文件通常是`/vmlinuz`、`/boot/vmlinuz`、`/bzImage` 或 `/boot/bzImage`。使用新內核前，請保存舊映像的副本，並複製新映像覆蓋舊映像。然後您【必須重新運行 LILO】來更新加載映射！否則，將無法啓動新的內核映像。

重新安裝 LILO 通常需要運行`/sbin/LILO`。您可能希望編輯`/etc/lilo.conf` 文件為舊內核映像指定一個條目（例如`vmlinuz.old`）防止新的不能正常工作。有關更多信息，請參閱 LILO 文檔。

重新安裝 LILO 之後，您應該就已經準備好了。關閉系統，重新啓動，盡情享受吧！

如果需要更改內核映像中的默認根設備、視頻模式等，請在適當的地方使用啟動裝載程序的引導選項。無需重新編譯內核即可更改這些參數。

- 使用新內核重新啓動並享受它吧。

## 若遇到問題

- 如果您發現了一些可能由於內核缺陷所導致的問題，請檢查 `MAINTAINERS`（維護者）文件看看是否有人與令您遇到麻煩的內核部分相關。如果無人在此列出，那麼第二個最好的方案就是把它們發給我（[torvalds@linux-foundation.org](mailto:torvalds@linux-foundation.org)），也可能發送到任何其他相關的郵件列表或新聞組。
- 在所有的缺陷報告中，【請】告訴我們您在說什麼內核，如何復現問題，以及您的設置是什麼的（使用您的常識）。如果問題是新的，請告訴我；如果問題是舊的，請嘗試告訴我您什麼時候首次注意到它。
- 如果缺陷導致如下消息：

```
unable to handle kernel paging request at address C0000010
Oops: 0002
```

```
EIP: 0010:XXXXXXXX
eax: XXXXXXXX ebx: XXXXXXXX ecx: XXXXXXXX edx: XXXXXXXX
esi: XXXXXXXX edi: XXXXXXXX ebp: XXXXXXXX
ds: xxxx es: xxxx fs: xxxx gs: xxxx
Pid: xx, process nr: xx
xx xx xx xx xx xx xx xx xx xx
```

或者類似的內核調試信息顯示在屏幕上或在系統日誌里，請【如實】複製它。可能對你來說轉儲 (dump) 看起來不可理解，但它確實包含可能有助於調試問題的信息。轉儲上方的文本也很重要：它說明了內核轉儲代碼的原因（在上面的示例中，是由於內核指針錯誤）。更多關於如何理解轉儲的信息，請參見 Documentation/admin-guide/bug-hunting.rst。

- 如果使用 CONFIG\_KALLSYMS 編譯內核，則可以按原樣發送轉儲，否則必須使用 `ksymoops` 程序來理解轉儲（但通常首選使用 CONFIG\_KALLSYMS 編譯）。此實用程序可從 <https://www.kernel.org/pub/linux/utils/kernel/ksymoops/> 下載。或者，您可以手動執行轉儲查找：
- 在調試像上面這樣的轉儲時，如果您可以查找 EIP 值的含義，這將非常有幫助。十六進位值本身對我或其他任何人都沒有太大幫助：它會取決於特定的內核設置。您應該做的是從 EIP 行獲取十六進位值（忽略 0010:），然後在內核名字列表中查找它，以查看哪個內核函數包含有問題的地址。

要找到內核函數名，您需要找到與顯示症狀的內核相關聯的系統二進位文件。就是文件「linux/vmlinux」。要提取名字列表並將其與內核崩潰中的 EIP 進行匹配，請執行：

```
nm vmlinux | sort | less
```

這將為您提供一個按升序排序的內核地址列表，從中很容易找到包含有問題的地址的函數。請注意，內核調試消息提供的地址不一定與函數地址完全匹配（事實上，這是不可能的），因此您不能只「grep」列表：不過列表將為您提供每個內核函數的起點，因此通過查找起始地址低於你正在搜索的地址，但後一個函數的高於的函數，你會找到您想要的。實際上，在您的問題報告中加入一些「上下文」可能是一個好主意，給出相關的上下幾行。

如果您由於某些原因無法完成上述操作（如您使用預編譯的內核映像或類似的映像），請儘可能多地告訴我您的相關設置信息，這會有所幫助。有關詳細信息請閱讀『Documentation/admin-guide/reporting-issues.rst』。

- 或者，您可以在正在運行的內核上使用 `gdb`（只讀的；即不能更改值或設置斷點）。為此，請首先使用`-g` 編譯內核；適當地編輯 `arch/x86/Makefile`，然後執行 `make clean`。您還需要啓用 `CONFIG_PROC_FS`（通過 `make config`）。

使用新內核重新啟動後，執行 `gdb vmlinux /proc/kcore`。現在可以使用所有普通的 `gdb` 命令。查找系統崩潰點的命令是 `l *0XXXXXXXX`（將 `xxx` 替換為 EIP 值）。

用 `gdb` 無法調試一個當前未運行的內核是由於 `gdb`（錯誤地）忽略了編譯內核的起始偏移量。

Todolist:

kernel-parameters devices sysctl/index

本節介紹 CPU 漏洞及其緩解措施。

Todolist:

hw-vuln/index

下面的一組文檔，針對的是試圖跟蹤問題和 bug 的用戶。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

**Original** Documentation/admin-guide/reporting-issues.rst

譯者 吳 想 成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)> 胡 皓 文 Hu Haowen  
[<src.res@email.cn>](mailto:<src.res@email.cn>)

## 報告問題

### 簡明指南（亦即太長不看）

您面臨的是否為同系列穩定版或長期支持內核的普通內核的回歸？是否仍然受支持？請搜索 LKML 內核郵件列表 和 Linux 穩定版郵件列表 存檔中匹配的報告並加入討論。如果找不到匹配的報告，請安裝該系列的最新版本。如果它仍然出現問題，報告給穩定版郵件列表 ([stable@vger.kernel.org](mailto:stable@vger.kernel.org))。

在所有其他情況下，請儘可能猜測是哪個內核部分導致了問題。查看 MAINTAINERS 文件，了解開發人員希望如何得知問題，大多數情況下，報告問題都是通過電子郵件和抄送相關郵件列表進行的。檢查報告目的地的存檔中是否已有匹配的報告；也請搜索 LKML 和網絡。如果找不到可加入的討論，請安裝 最新的主線內核。如果仍存在問題，請發送報告。

問題已經解決了，但是您希望看到它在一個仍然支持的穩定版或長期支持系列中得到解決？請安裝其最新版本。如果它出現了問題，那麼在主線中搜索修復它的更改，並檢查是否正在回傳（backporting）或者已放棄；如果兩者都沒有，那麼可詢問處理更改的人員。

**通用提醒：**當安裝和測試上述內核時，請確保它是普通的（即：沒有補丁，也沒有使用附加模塊）。還要確保它是在一個正常的環境中構建和運行，並且在問題發生之前沒有被汙染（tainted）。

在編寫報告時，要涵蓋與問題相關的所有信息，如使用的內核和發行版。在碰見回歸時，嘗試給出引入它的更改的提交 ID，二分可以找到它。如果您同時面臨 Linux 內核的多個問題，請分別報告每個問題。

一旦報告發出，請回答任何出現的問題，並儘可能地提供幫助。這包括通過不時重新測試新版本並發送狀態更新來推動進展。

### 如何向內核維護人員報告問題的逐步指南

上面的簡明指南概述了如何向 Linux 內核開發人員報告問題。對於已經熟悉向自由和開源軟體（FLOSS）項目報告問題的人來說，這可能是他們所需要的全部內容。對於其他人，本部分更為詳細，並一步一步地描述。為了便於閱讀，它仍然儘量簡潔，並省略了許多細節；這些在逐步指南後的參考章節中進行了描述，該章節更詳細地解釋了每個步驟。

注意：本節涉及的方面比簡明指南多，順序也稍有不同。這符合你的利益，以確保您儘早意識到看起來像 Linux 內核毛病的問題可能實際上是由其他原因引起的。這些步驟可以確保你最終不會覺得在這一過程中投入的時間是浪費：

- 您是否面臨硬體或軟體供應商提供的 Linux 內核的問題？那麼基本上您最好停止閱讀本文檔，轉而向您的供應商報告問題，除非您願意自己安裝最新的 Linux 版本。尋找和解決問題往往需要後者。
- 使用您喜愛的網絡搜尋引擎對現有報告進行粗略搜索；此外，請檢查 [Linux 內核郵件列表（LKML）](#) 的存檔。如果找到匹配的報告，請加入討論而不是發送新報告。
- 看看你正在處理的問題是否為回歸問題、安全問題或非常嚴重的問題：這些都是需要在接下來的一些步驟中特別處理的「高優先級問題」。
- 確保不是內核環境導致了您面臨的問題。
- 創建一個新的備份，並將系統修復和恢復工具放在手邊。
- 確保您的系統不會通過動態構建額外的內核模塊來增強其內核，像 DKMS 這樣的解決方案可能在您不知情的情況下就在本地進行了這樣的工作。
- 當問題發生時，檢查您的內核是否被「汙染」，因為使內核設置這個標誌的事件可能會導致您面臨的問題。
- 粗略地寫下如何重現這個問題。如果您同時處理多個問題，請為每個問題單獨寫注釋，並確保它們在新啓動的系統上獨立出現。這是必要的，因為每個問題都需要分別報告給內核開發人員，除非它們嚴重糾纏在一起。
- 如果您正面臨穩定版或長期支持版本線的回歸（例如從 5.10.4 更新到 5.10.5 時出現故障），請查看後文「報告穩定版和長期支持內核線的回歸」小節。
- 定位可能引起問題的驅動程序或內核子系統。找出其開發人員期望的報告的方式和位置。注意：大多數情況下不會是 [bugzilla.kernel.org](https://bugzilla.kernel.org)，因為問題通常需要通過郵件發送給維護人員和公共郵件列表。
- 在缺陷追蹤器或問題相關郵件列表的存檔中徹底搜索可能與您的問題匹配的報告。如果你發現了一些相關討論，請加入討論而不是發送新的報告。

在完成這些準備之後，你將進入主要部分：

- 除非您已經在運行最新的「主線」Linux 內核，否則最好在報告流程前安裝它。在某些情況下，使用最新的「穩定版」Linux 進行測試和報告也是可以接受的替代方案；在合併窗口期間，這實際上可能是最好的方法，但在開發階段最好還是暫停幾天。無論你選擇什麼版本，最好使用「普通」構建。忽略這些建議會大大增加您的報告被拒絕或忽略的風險。
- 確保您剛剛安裝的內核在運行時不會「汙染」自己。

- 在您剛剛安裝的內核中復現這個問題。如果它沒有出現，請查看下方只發生在穩定版和長期支持內核的問題的說明。
- 優化你的筆記：試著找到並寫出最直接的復現問題的方法。確保最終結果包含所有重要的細節，同時讓第一次聽說的人容易閱讀和理解。如果您在此過程中學到了一些東西，請考慮再次搜索關於該問題的現有報告。
- 如果失敗涉及「panic」、「Oops」、「warning」或「BUG」，請考慮解碼內核日誌以查找觸發錯誤的代碼行。
- 如果您的問題是回歸問題，請儘可能縮小引入問題時的範圍。
- 通過詳細描述問題來開始編寫報告。記得包括以下條目：您為復現而安裝的最新內核版本、使用的 Linux 發行版以及關於如何復現該問題的說明。如果可能，將內核構建配置 (.config) 和 dmesg 的輸出放在網上的某個地方，並連結到它。包含或上傳所有其他可能相關的信息，如 Oops 的輸出/截圖或來自 lspci 的輸出。一旦你寫完了這個主要部分，請在上方插入一個正常長度的段落快速概述問題和影響。再在此之上添加一個簡單描述問題的句子，以得到人們的閱讀。現在給出一個更短的描述性標題或主題。然後就可以像 MAINTAINERS 文件告訴你的那樣發送或提交報告了，除非你在處理一個「高優先級問題」：它們需要按照下面「高優先級問題的特殊處理」所述特別關照。
- 等待別人的反應，繼續推進事情，直到你能夠接受這樣或那樣的結果。因此，請公開和及時地回應任何詢問。測試提出的修復。積極地測試：至少重新測試每個新主線版本的首個候選版本 (RC)，並報告你的結果。如果出現拖延，就友好地提醒一下。如果你沒有得到任何幫助或者未能滿意，請試著自己幫助自己。

## 報告穩定版和長期支持內核線的回歸

如果您發現了穩定版或長期支持內核版本線中的回歸問題並按上述流程跳到這裡，那麼請閱讀本小節。即例如您在從 5.10.4 更新到 5.10.5 時出現了問題（從 5.9.15 到 5.10.5 則不是）。開發人員希望儘快修復此類回歸，因此有一個簡化流程來報告它們：

- 檢查內核開發人員是否仍然維護你關心的 Linux 內核版本線：去 [kernel.org](http://kernel.org) 的首頁，確保此特定版本線的最新版沒有「[EOL]」標記。
- 檢查 [Linux 穩穩版郵件列表](#) 中的現有報告。
- 從特定的版本線安裝最新版本作為純淨內核。確保這個內核沒有被汙染，並且仍然存在問題，因為問題可能已經在那裡被修復了。如果您第一次發現供應商內核的問題，請檢查已知最新版本的普通構建是否可以正常運行。
- 向 [Linux 穩穩版郵件列表](mailto:stable@vger.kernel.org) 發送一個簡短的問題報告 (stable@vger.kernel.org)。大致描述問題，並解釋如何復現。講清楚首個出現問題的版本和最後一個工作正常的版本。然後等待進一步的指示。

下面的參考章節部分詳細解釋了這些步驟中的每一步。

### 報告只發生在較舊內核版本線的問題

若您嘗試了上述的最新主線內核，但未能在那裡復現問題，那麼本小節適用於您；以下流程有助於使問題在仍然支持的穩定版或長期支持版本線，或者定期基於最新穩定版或長期支持內核的供應商內核中得到修復。如果是這種情況，請執行以下步驟：

- 請做好準備，接下來的幾個步驟可能無法在舊版本中解決問題：修復可能太大或太冒險，無法移植到那裡。
- 執行前節「報告穩定版和長期支持內核線的回歸」中的前三個步驟。
- 在 Linux 內核版本控制系統中搜索修復主線問題的更改，因為它的提交消息可能會告訴你修復是否已經計劃好了支持。如果你沒有找到，搜索適當的郵件列表，尋找討論此類問題或同行評議可能修復的帖子；然後檢查討論是否認為修復不適合支持。如果支持根本不被考慮，加入最新的討論，詢問是否有可能。
- 前面的步驟之一應該會給出一個解決方案。如果仍未能成功，請向可能引起問題的子系統的維護人員詢問建議；抄送特定子系統的郵件列表以及穩定版郵件列表

下面的參考章節部分詳細解釋了這些步驟中的每一步。

### 參考章節：向內核維護者報告問題

上面的詳細指南簡要地列出了所有主要步驟，這對大多數人來說應該足夠了。但有時，即使是有經驗的用戶也可能想知道如何實際執行這些步驟之一。這就是本節的目的，因為它將提供關於上述每個步驟的更多細節。請將此作為參考文檔：可以從頭到尾閱讀它。但它主要是為了瀏覽和查找如何實際執行這些步驟的詳細信息。

在深入挖掘細節之前，我想先給你一些一般性建議：

- Linux 內核開發人員很清楚這個過程很複雜，比其他的 FLOSS 項目要求更多。我們很希望讓它更簡單。但這需要在不同的地方以及一些基礎設施上付諸努力，這些基礎設施需要持續的維護；尚未有人站出來做這些工作，所以目前情況就是這樣。
- 與某些供應商簽訂的保證或支持合同並不能使您有權要求上游 Linux 內核社區的開發人員進行修復：這樣的合同完全在 Linux 內核、其開發社區和本文檔的範圍之外。這就是為什麼在這種情況下，你不能要求任何契約保證，即使開發人員處理的問題對供應商有效。如果您想主張您的權利，使用供應商的支持渠道代替。當這樣做的時候，你可能想提出你希望看到這個問題在上游 Linux 內核中修復；可以這是確保最終修復將被納入所有 Linux 發行版的唯一方法來鼓勵他們。
- 如果您從未向 FLOSS 項目報告過任何問題，那麼您應該考慮閱讀 [如何有效地報告缺陷](#)，如何以明智的方式提問，和 [如何提出好問題](#)。

解決這些問題之後，可以在下面找到如何正確地向 Linux 內核報告問題的詳細信息。

## 確保您使用的是上游 Linux 內核

您是否面臨硬體或軟體供應商提供的 Linux 內核的問題？那麼基本上您最好停止閱讀本文檔，轉而向您的供應商報告問題，除非您願意自己安裝最新的 Linux 版本。尋找和解決問題往往需要後者。

與大多數程式設計師一樣，Linux 內核開發人員不喜歡花時間處理他們維護的原始碼中根本不會發生的問題的報告。這只會浪費每個人的時間，尤其是你的時間。不幸的是，當涉及到內核時，這樣的情況很容易發生，並且常常導致雙方氣餒。這是因為幾乎所有預裝在設備（台式機、筆記本電腦、智慧型手機、路由器等）上的 Linux 內核，以及大多數由 Linux 發行商提供的內核，都與由 kernel.org 發行的官方 Linux 內核相距甚遠：從 Linux 開發的角度來看，這些供應商提供的內核通常是古老的或者經過了大量修改，通常兩點兼具。

大多數供應商內核都不適合用來向 Linux 內核開發人員報告問題：您在其中遇到的問題可能已經由 Linux 內核開發人員在數月或數年前修復；此外，供應商的修改和增強可能會導致您面臨的問題，即使它們看起來很小或者完全不相關。這就是為什麼您應該向供應商報告這些內核的問題。它的開發者應該查看報告，如果它是一個上游問題，直接於上游修復或將報告轉發到那裡。在實踐中，這有時行不通。因此，您可能需要考慮通過自己安裝最新的 Linux 內核來繞過供應商。如果如果您選擇此方法，那麼本指南後面的步驟將解釋如何在排除了其他可能導致您的問題的原因後執行此操作。

注意前段使用的詞語是「大多數」，因為有時候開發人員實際上願意處理供應商內核出現的問題報告。他們是否這麼做很大程度上取決於開發人員和相關問題。如果發行版只根據最近的 Linux 版本對內核進行了較小修改，那麼機會就比較大；例如對於 Debian GNU/Linux Sid 或 Fedora Rawhide 所提供的主線內核。一些開發人員還將接受基於最新穩定內核的發行版內核問題報告，只要它改動不大；例如 Arch Linux、常規 Fedora 版本和 openSUSE Turboweed。但是請記住，您最好使用主線 Linux，並避免在此流程中使用穩定版內核，如「安裝一個新的內核進行測試」一節中所詳述。

當然，您可以忽略所有這些建議，並向上游 Linux 開發人員報告舊的或經過大量修改的供應商內核的問題。但是注意，這樣的報告經常被拒絕或忽視，所以自行小心考慮一下。不過這還是比根本不報告問題要好：有時候這樣的報告會直接或間接地幫助解決之後的問題。

### 搜索現有報告（第一部分）

使用您喜愛的網絡搜尋引擎對現有報告進行粗略搜索；此外，請檢查 Linux 內核郵件列表（LKML）的存檔。如果找到匹配的報告，請加入討論而不是發送新報告。

報告一個別人已經提出的問題，對每個人來說都是浪費時間，尤其是作為報告人的你。所以徹底檢查是否有人已經報告了這個問題，這對你自己是有利的。在流程中的這一步，可以只執行一個粗略的搜索：一旦您知道您的問題需要報告到哪裡，稍後的步驟將告訴您如何詳細搜索。儘管如此，不要倉促完成這一步，它可以節省您的時間和減少麻煩。

只需先用你最喜歡的搜尋引擎在網際網路上搜索。然後再搜索 Linux 內核郵件列表（LKML）存檔。

如果搜索結果實在太多，可以考慮讓你的搜尋引擎將搜索時間範圍限制在過去的一個月或一年。而且無論你在哪里搜索，一定要用恰當的搜索關鍵詞；也要變化幾次關鍵詞。同時，試著從別人的角度看問題：這將幫助你想出其他的關鍵詞。另外，一定不要同時使用過多的關鍵詞。記住搜索時要同時嘗試包含和不包含內核驅動程序的名稱或受影響的硬體組件的名稱等信息。但其確切的品牌名稱（比如說「華碩紅魔 Radeon RX

5700 XT Gaming OC」) 往往幫助不大，因為它太具體了。相反，嘗試搜索術語，如型號 (Radeon 5700 或 Radeon 5000) 和核心代號 (「Navi」或「Navi10」)，以及包含和不包含其製造商 (「AMD」)。

如果你發現了關於你的問題的現有報告，請加入討論，因為你可能會提供有價值的額外信息。這一點很重要，即使是在修復程序已經準備好或處於最後階段，因為開發人員可能會尋找能夠提供額外信息或測試建議修復程序的人。跳到「發布報告後的責任」一節，了解有關如何正確參與的細節。

注意，搜索 [bugzilla.kernel.org](https://bugzilla.kernel.org) 網站可能也是一個好主意，因為這可能會提供有價值的見解或找到匹配的報告。如果您發現後者，請記住：大多數子系統都希望在不同的位置報告，如下面「你需要將問題報告到何處」一節中所述。因此本應處理這個問題的開發人員甚至可能不知道 bugzilla 的工單。所以請檢查工單中的問題是否已經按照本文檔所述得到報告，如果沒有，請考慮這樣做。

### 高優先級的問題？

看看你正在處理的問題是否是回歸問題、安全問題或非常嚴重的問題：這些都是需要在接下來的一些步驟中特別處理的「高優先級問題」。

Linus Torvalds 和主要的 Linux 內核開發人員希望看到一些問題儘快得到解決，因此在報告過程中有一些「高優先級問題」的處理略有不同。有三種情況符合條件：回歸、安全問題和非常嚴重的問題。

如果在舊版本的 Linux 內核中工作的東西不能在新版本的 Linux 內核中工作，或者某種程度上在新版本的 Linux 內核中工作得更差，那麼你就需要處理「回歸」。因此，當一個在 Linux 5.7 中表現良好的 WiFi 驅動程序在 5.8 中表現不佳或根本不能工作時，這是一種回歸。如果應用程式在新的內核中出現不穩定的現象，這也是一種回歸，這可能是由於內核和用戶空間之間的接口（如 procfs 和 sysfs）發生不兼容的更改造成的。顯著的性能降低或功耗增加也可以稱為回歸。但是請記住：新內核需要使用與舊內核相似的配置來構建（參見下面如何實現這一點）。這是因為內核開發人員在實現新特性時有時無法避免不兼容性；但是為了避免回歸，這些特性必須在構建配置期間顯式地啓用。

什麼是安全問題留給你自己判斷。在繼續之前，請考慮閱讀「安全缺陷」，因為它提供了如何最恰當地處理安全問題的額外細節。

當發生了完全無法接受的糟糕事情時，此問題就是一個「非常嚴重的問題」。例如，Linux 內核破壞了它處理的數據或損壞了它運行的硬體。當內核突然顯示錯誤消息 (「kernel panic」) 並停止工作，或者根本沒有任何停止信息時，您也在處理一個嚴重的問題。注意：不要混淆「panic」（內核停止自身的致命錯誤）和「Oops」（可恢復錯誤），因為顯示後者之後內核仍然在運行。

### 確保環境健康

確保不是內核所處環境導致了你所面臨的問題。

看起來很像內核問題的問題有時是由構建或運行時環境引起的。很難完全排除這種問題，但你應該儘量減少這種問題：

- 構建內核時，請使用經過驗證的工具，因為編譯器或二進位文件中的錯誤可能會導致內核出現錯誤行為。

- 確保您的計算機組件在其設計規範內運行；這對處理器、內存和主板尤為重要。因此，當面臨潛在的內核問題時，停止低電壓或超頻。
- 儘量確保不是硬體故障導致了你的問題。例如，內存損壞會導致大量的問題，這些問題會表現為看起來像內核問題。
- 如果你正在處理一個文件系統問題，你可能需要用 `fsck` 檢查一下文件系統，因為它可能會以某種方式被損壞，從而導致無法預期的內核行為。
- 在處理回歸問題時，要確保沒有在更新內核的同時發生了其他變化。例如，這個問題可能是由同時更新的其他軟體引起的。也有可能是在你第一次重啓進入新內核時，某個硬體巧合地壞了。更新系統 BIOS 或改變 BIOS 設置中的某些內容也會導致一些看起來很像內核回歸的問題。

## 為緊急情況做好準備

創建一個全新的備份，並將系統修復和還原工具放在手邊

我得提醒您，您正在和計算機打交道，計算機會出現意想不到的事情，尤其是當您拆騰其作業系統的內核等關鍵部件時。而這就是你在這個過程中要做的事情。因此，一定要創建一個全新的備份；還要確保你手頭有修復或重裝作業系統的所有工具，以及恢復備份所需的一切。

## 確保你的內核不會被增強

確保您的系統不會通過動態構建額外的內核模塊來增強其內核，像 `DKMS` 這樣的解決方案可能在您不知情的情況下就在本地進行了這樣的工作。

如果內核以任何方式得到增強，那麼問題報告被忽略或拒絕的風險就會急劇增加。這就是為什麼您應該刪除或禁用像 `akmods` 和 `DKMS` 這樣的機制：這些機制會自動構建額外內核模塊，例如當您安裝新的 Linux 內核或第一次引導它時。也要記得同時刪除他們可能安裝的任何模塊。然後重新啓動再繼續。

注意，你可能不知道你的系統正在使用這些解決方案之一：當你安裝 Nvidia 專有圖形驅動程序、Virtual-Box 或其他需要 Linux 內核以外的模塊支持的軟體時，它們通常會靜默設置。這就是為什麼你可能需要卸載這些軟體的軟體包，以擺脫任何第三方內核模塊。

## 檢測「汙染」標誌

當問題發生時，檢查您的內核是否被「汙染」，因為使內核設置這個標誌的事件可能會導致您面臨的問題。

當某些可能會導致看起來完全不相關的後續錯誤的事情發生時，內核會用「汙染 (taint)」標誌標記自己。如果您的內核受到汙染，那麼您面臨的可能是這樣的錯誤。因此在投入更多時間到這個過程中之前，儘早排除此情況可能對你有好處。這是這個步驟出現在這裡的唯一原因，因為這個過程稍後會告訴您安裝最新的主線內核；然後您將需要再次檢查汙染標誌，因為當它出問題的時候內核報告會關注它。

在正在運行的系統上檢查內核是否汙染非常容易：如果 `cat /proc/sys/kernel/tainted` 返回「0」，那麼內核沒有被汙染，一切正常。在某些情況下無法檢查該文件；這就是為什麼當內核報告內部問題（「kernel

bug」)、可恢復錯誤 (「kernel Oops」) 或停止操作前不可恢復的錯誤 (「kernel panic」) 時，它也會提到汙染狀態。當其中一個錯誤發生時，查看列印的錯誤消息的頂部，搜索以「CPU:」開頭的行。如果發現問題時內核未被汙染，那麼它應該以「Not infected」結束；如果你看到「Tainted:」且後跟一些空格和字母，那就被汙染了。

如果你的內核被汙染了，請閱讀「[受汙染的內核](#)」以找出原因。設法消除汙染因素。通常是由以下三種因素之一引起的：

1. 發生了一個可恢復的錯誤 (「kernel Oops」)，內核汙染了自己，因為內核知道在此之後它可能會出現奇怪的行為錯亂。在這種情況下，檢查您的內核或系統日誌，並尋找以下列文字開頭的部分：

```
Oops: 0000 [#1] SMP
```

如方括號中的「#1」所示，這是自啓動以來的第一次 Oops。每個 Oops 和此後發生的任何其他問題都可能是首個 Oops 的後續問題，即使這兩個問題看起來完全不相關。通過消除首個 Oops 的原因並在之後復現該問題，可以排除這種情況。有時僅僅重新啓動就足夠了，有時更改配置後重新啓動可以消除 Oops。但是在這個流程中不要花費太多時間在這一點上，因為引起 Oops 的原因可能已經在您稍後將按流程安裝的新 Linux 內核版本中修復了。

2. 您的系統使用的軟體安裝了自己的內核模塊，例如 Nvidia 的專有圖形驅動程序或 VirtualBox。當內核從外部源（即使它們是開源的）加載此類模塊時，它會汙染自己：它們有時會在不相關的內核區域導致錯誤，從而可能導致您面臨的問題。因此，當您想要向 Linux 內核開發人員報告問題時，您必須阻止這些模塊加載。大多數情況下最簡單的方法是：臨時卸載這些軟體，包括它們可能已經安裝的任何模塊。之後重新啓動。
3. 當內核加載駐留在 Linux 內核原始碼 staging 樹中的模塊時，它也會汙染自身。這是一個特殊的區域，代碼（主要是驅動程序）還沒有達到正常 Linux 內核的質量標準。當您報告此種模塊的問題時，內核受到汙染顯然是沒有問題的；只需確保問題模塊是造成汙染的唯一原因。如果問題發生在一個不相關的區域，重新啓動並通過指定 `foo.blacklist=1` 作為內核參數臨時阻止該模塊被加載（用有問題的模塊名替換「`foo`」）。

### 記錄如何重現問題

粗略地寫下如何重現這個問題。如果您同時處理多個問題，請為每個問題單獨寫注釋，並確保它們在新啓動的系統上獨立出現。這是必要的，因為每個問題都需要分別報告給內核開發人員，除非它們嚴重糾纏在一起。

如果你同時處理多個問題，必須分別報告每個問題，因為它們可能由不同的開發人員處理。在一份報告中描述多種問題，也會讓其他人難以將其分開。因此只有在問題嚴重糾纏的情況下，才能將問題合併在一份報告中。

此外，在報告過程中，你必須測試該問題是否發生在其他內核版本上。因此，如果您知道如何在一個新啓動的系統上快速重現問題，將使您的工作更加輕鬆。

注意：報告只發生過一次的問題往往是沒有結果的，因為它們可能是由於宇宙輻射導致的位翻轉。所以你應該嘗試通過重現問題來排除這種情況，然後再繼續。如果你有足夠的經驗來區分由於硬體故障引起的一次性錯誤和難以重現的罕見內核問題，可以忽略這個建議。

## 穩定版或長期支持內核的回歸？

如果您正面臨穩定版或長期支持版本線的回歸（例如從 5.10.4 更新到 5.10.5 時出現故障），請查看後文「報告穩定版和長期支持內核線的回歸」小節。

穩定版和長期支持內核版本線中的回歸是 Linux 開發人員非常希望解決的問題，這樣的問題甚至比主線開發分支中的回歸更不應出現，因為它們會很快影響到很多人。開發人員希望儘快了解此類問題，因此有一個簡化流程來報告這些問題。注意，使用更新內核版本線的回歸（比如從 5.9.15 切換到 5.10.5 時出現故障）不符合條件。

## 你需要將問題報告到何處

定位可能引起問題的驅動程序或內核子系統。找出其開發人員期望的報告的方式和位置。注意：大多數情況下不會是 [bugzilla.kernel.org](https://bugzilla.kernel.org)，因為問題通常需要通過郵件發送給維護人員和公共郵件列表。

將報告發送給合適的人是至關重要的，因為 Linux 內核是一個大項目，大多數開發人員只熟悉其中的一小部分。例如，相當多的程式設計師只關心一個驅動程序，比如一個 WiFi 晶片驅動程序；它的開發人員可能對疏遠的或不相關的「子系統」（如 TCP 堆棧、PCIe/PCI 子系統、內存管理或文件系統）的內部知識了解很少或完全不了解。

問題在於：Linux 內核缺少一個，可以簡單地將問題歸檔並讓需要了解它的開發人員了解它的，中心化缺陷跟蹤器。這就是為什麼你必須找到正確的途徑來自己報告問題。您可以在腳本的幫助下做到這一點（見下文），但它主要針對的是內核開發人員和專家。對於其他人來說，MAINTAINERS（維護人員）文件是更好的選擇。

## 如何閱讀 MAINTAINERS 維護者文件

為了說明如何使用 MAINTAINERS 文件，讓我們假設您的筆記本電腦中的 WiFi 在更新內核後突然出現了錯誤行為。這種情況下可能是 WiFi 驅動的問題。顯然，它也可能由於驅動基於的某些代碼，但除非你懷疑有這樣的東西會附著在驅動程序上。如果真的是其他的問題，驅動程序的開發人員會讓合適的人參與進來。

遺憾的是，沒有通用且簡單的辦法來檢查哪個代碼驅動了特定硬體組件。

在 WiFi 驅動出現問題的情況下，你可能想查看 `lspci -k` 的輸出，因為它列出了 PCI/PCIe 總線上的設備和驅動它的內核模塊：

```
[user@something ~]$ lspci -k
[...]
3a:00.0 Network controller: Qualcomm Atheros QCA6174 802.11ac Wireless Network Adapter (rev 32)
    Subsystem: Bigfoot Networks, Inc. Device 1535
    Kernel driver in use: ath10k_pci
    Kernel modules: ath10k_pci
[...]
```

但如果您的 WiFi 晶片通過 USB 或其他內部總線連接，這種方法就行不通了。在這種情況下，您可能需要檢查您的 WiFi 管理器或 `ip link` 的輸出。尋找有問題的網絡接口的名稱，它可能類似於「`wlp58s0`」。此名稱

可以用來找到驅動它的模塊:

```
[user@something ~]$ realpath --relative-to=/sys/module//sys/class/net/wlp58s0/device/driver/  
└─module  
ath10k_pci
```

如果這些技巧不能進一步幫助您，請嘗試在網上搜索如何縮小相關驅動程序或子系統的範圍。如果你不確定是哪一個：試著猜一下，即使你猜得不好，也會有人會幫助你的。

一旦您知道了相應的驅動程序或子系統，您就希望在 MAINTAINERS 文件中搜索它。如果是「`ath10k_pci`」，您不會找到任何東西，因為名稱太具體了。有時你需要在網上尋找幫助；但在此之前，請嘗試使用一個稍短或修改過的名稱來搜索 MAINTAINERS 文件，因為這樣你可能會發現類似這樣的東西：

```
QUALCOMM AHEROS ATH10K WIRELESS DRIVER  
Mail: A. Some Human <shuman@example.com>  
Mailing list: ath10k@lists.infradead.org  
Status: Supported  
Web-page: https://wireless.wiki.kernel.org/en/users/Drivers/ath10k  
SCM: git git://git.kernel.org/pub/scm/linux/kernel/git/kvalo/ath.git  
Files: drivers/net/wireless/ath/ath10k/
```

注意：如果您閱讀在 Linux 原始碼樹的根目錄中找到的原始維護者文件，則行描述將是縮寫。例如，「`Mail:` (郵件)」將是「`M:`」，「`Mailing list:` (郵件列表)」將是「`L`」，「`Status:` (狀態)」將是「`S:`」。此文件頂部有一段解釋了這些和其他縮寫。

首先查看「`Status`」狀態行。理想情況下，它應該得到「`Supported` (支持)」或「`Maintained` (維護)」。如果狀態為「`Obsolete` (過時的)」，那麼你在使用一些過時的方法，需要轉換到新的解決方案上。有時候，只有在感到有動力時，才會有人為代碼提供「`Odd Fixes`」。如果碰見「`Orphan`」，你就完全不走運了，因為再也沒有人關心代碼了，只剩下這些選項：準備好與問題共存，自己修復它，或者找一個願意修復它的程式設計師。

檢查狀態後，尋找以「`bug:`」開頭的一行：它將告訴你在哪裡可以找到子系統特定的缺陷跟蹤器來提交你的問題。上面的例子沒有此行。大多數部分都是這樣，因為 Linux 內核的開發完全是由郵件驅動的。很少有子系統使用缺陷跟蹤器，且其中只有一部分依賴於 `bugzilla.kernel.org`。

在這種以及其他很多情況下，你必須尋找以「`Mail:`」開頭的行。這些行提到了特定代碼的維護者的名字和電子郵件地址。也可以查找以「`Mailing list:`」開頭的行，它告訴你開發代碼的公共郵件列表。你的報告之後需要通過郵件發到這些地址。另外，對於所有通過電子郵件發送的問題報告，一定要抄送 Linux Kernel Mailing List (LKML) <[linux-kernel@vger.kernel.org](mailto:linux-kernel@vger.kernel.org)>。在以後通過郵件發送問題報告時，不要遺漏任何一個郵件列表！維護者都是大忙人，可能會把一些工作留給子系統特定列表上的其他開發者；而 LKML 很重要，因為需要一個可以找到所有問題報告的地方。

## 藉助腳本找到維護者

對於手頭有 Linux 源碼的人來說，有第二個可以找到合適的報告地點的選擇：腳本「scripts/get\_maintainer.pl」，它嘗試找到所有要聯繫的人。它會查詢 MAINTAINERS 文件，並需要用相關原始碼的路徑來調用。對於編譯成模塊的驅動程序，經常可以用這樣的命令找到：

```
$ modinfo ath10k_pci | grep filename | sed 's!/lib/modules/.*/kernel/!!; s!filename:!!; s!\.ko\
˓→(\|\.xz\)\!!'
drivers/net/wireless/ath/ath10k/ath10k_pci.ko
```

將其中的部分內容傳遞給腳本：

```
$ ./scripts/get_maintainer.pl -f drivers/net/wireless/ath/ath10k*
Some Human <shuman@example.com> (supporter:QUALCOMM Atheros ATH10K WIRELESS DRIVER)
Another S. Human <asomehuman@example.com> (maintainer:NETWORKING DRIVERS)
ath10k@lists.infradead.org (open list:QUALCOMM Atheros ATH10K WIRELESS DRIVER)
linux-wireless@vger.kernel.org (open list:NETWORKING DRIVERS (WIRELESS))
netdev@vger.kernel.org (open list:NETWORKING DRIVERS)
linux-kernel@vger.kernel.org (open list)
```

不要把你的報告發給所有的人。發送給維護者，腳本稱之為「supporter:」；另外抄送代碼最相關的郵件列表，以及 Linux 內核郵件列表（LKML）。在此例中，你需要將報告發送給「Some Human <shuman@example.com>」，並抄送「ath10k@lists.infradead.org」和「linux-kernel@vger.kernel.org」。

注意：如果你用 git 克隆了 Linux 原始碼，你可能需要用–git 再次調用 get\_maintainer.pl。腳本會查看提交歷史，以找到最近哪些人參與了相關代碼的編寫，因為他們可能會提供幫助。但要小心使用這些結果，因為它很容易讓你誤入歧途。例如，這種情況常常會發生在很少被修改的地方（比如老舊的或未維護的驅動程序）：有時這樣的代碼會在樹級清理期間被根本不關心此驅動程序的開發者修改。

## 搜索現有報告（第二部分）

在缺陷追蹤器或問題相關郵件列表的存檔中徹底搜索可能與您的問題匹配的報告。如果找到匹配的報告，請加入討論而不是發送新報告。

如前所述：報告一個別人已經提出的問題，對每個人來說都是浪費時間，尤其是作為報告人的你。這就是為什麼你應該再次搜索現有的報告。現在你已經知道問題需要報告到哪裡。如果是郵件列表，那麼一般在 [lore.kernel.org](#) 可以找到相應存檔。

但有些列表運行在其他地方。例如前面步驟中當例子的 ath10k WiFi 驅動程序就是這種情況。但是你通常可以在網上很容易地找到這些列表的檔案。例如搜索「archive [ath10k@lists.infradead.org](#)」，將引導您到 ath10k 郵件列表的信息頁，該頁面頂部連結到其 [列表存檔](#)。遺憾的是，這個列表和其他一些列表缺乏搜索其存檔的功能。在這種情況下可以使用常規的網際網路搜尋引擎，並添加類似「site:[lists.infradead.org/pipermail/ath10k/](#)」這樣的搜索條件，這會把結果限制在該連結中的檔案。

也請進一步搜索網絡、LKML 和 [bugzilla.kernel.org](#) 網站。

有關如何搜索以及在找到匹配報告時如何操作的詳細信息，請參閱上面的「[搜索現有報告（第一部分）](#)」。

不要急著完成報告過程的這一步：花 30 到 60 分鐘甚至更多的時間可以為你和其他人節省 / 減少相當多的時間和麻煩。

### 安裝一個新的內核進行測試

除非您已經在運行最新的「主線」Linux 內核，否則最好在報告流程前安裝它。在某些情況下，使用最新的「穩定版」Linux 進行測試和報告也是可以接受的替代方案；在合併窗口期間，這實際上可能是最好的方法，但在開發階段最好還是暫停幾天。無論您選擇什麼版本，最好使用「普通」構建。忽略這些建議會大大增加您的報告被拒絕或忽略的風險。

正如第一步的詳細解釋中所提到的：與大多數程式設計師一樣，與大多數程式設計師一樣，Linux 內核開發人員不喜歡花時間處理他們維護的原始碼中根本不會發生的問題的報告。這隻會浪費每個人的時間，尤其是你的時間。這就是為什麼在報告問題之前，您必須先確認問題仍然存在於最新的上游代碼中，這符合每個人的利益。您可以忽略此建議，但如前所述：這樣做會極大地增加問題報告被拒絕或被忽略的風險。

內核「最新上游」的範圍通常指：

- 安裝一個主線內核；最新的穩定版內核也可以是一個選擇，但大多數時候都最好避免。長期支持內核（有時稱為「LTS 內核」）不適合此流程。下一小節將更詳細地解釋所有這些。
- 下一小節描述獲取和安裝這樣一個內核的方法。它還指出了使用預編譯內核是可以的，但普通的內核更好，這意味著：它是直接使用從 [kernel.org](https://kernel.org) 獲得的 Linux 原始碼構建並且沒有任何方式修改或增強。

### 選擇適合測試的版本

前往 [kernel.org](https://kernel.org) 來決定使用哪個版本。忽略那個寫著「Latest release 最新版本」的巨大黃色按鈕，往下看有一個表格。在表格的頂部，你會看到一行以「mainline」開頭的字樣，大多數情況下它會指向一個版本號類似「5.8-rc2」的預發布版本。如果是這樣的話，你將需要使用這個主線內核進行測試。不要讓「rc」嚇到你，這些「開發版內核」實際上非常可靠——而且你已經按照上面的指示做了備份，不是嗎？

大概每九到十周，「mainline」可能會給你指出一個版本號類似「5.7」的正式版本。如果碰見這種情況，請考慮暫停報告過程，直到下一個版本的第一個預發布（5.8-rc1）出現在 [kernel.org](https://kernel.org) 上。這是因為 Linux 的開發周期正在兩周的「合併窗口」內。大部分的改動和所有干擾性的改動都會在這段時間內被合併到下一個版本中。在此期間使用主線是比較危險的。內核開發者通常也很忙，可能沒有多餘的時間來處理問題報告。這也是很有可能在合併窗口中應用了許多修改來修復你所面臨的問題；這就是為什麼你很快就得用一個新的內核版本重新測試，就像下面「[發布報告後的責任](#)」一節中所述的那樣。

這就是為什麼要等到合併窗口結束後才去做。但是如果你處理的是一些不應該等待的東西，則無需這樣做。在這種情況下，可以考慮通過 git 獲取最新的主線內核（見下文），或者使用 [kernel.org](https://kernel.org) 上提供的最新穩定版本。如果 mainline 因為某些原因不無法正常工作，那麼使用它也是可以接受的。總的來說：用它來重現問題也比完全不報告問題要好。

最好避免在合併窗口外使用最新的穩定版內核，因為所有修復都必須首先應用於主線。這就是為什麼檢查最新的主線內核是如此重要：你希望看到在舊版本線修復的任何問題需要先在主線修復，然後才能得到回傳，

這可能需要幾天或幾周。另一個原因是：您希望的修復對於回傳來說可能太難或太冒險；因此再次報告問題不太可能改變任何事情。

這些方面也部分表明了為什麼長期支持內核（有時稱為「LTS 內核」）不適合報告流程：它們與當前任碼的距離太遠。因此，先去測試主線，然後再按流程走：如果主線沒有出現問題，流程將指導您如何在舊版本線中修復它。

## 如何獲得新的 Linux 內核

你可以使用預編譯或自編譯的內核進行測試；如果你選擇後者，可以使用 `git` 獲取源代碼，或者下載其 tar 存檔包。

**使用預編譯的內核：**這往往是最快速、最簡單、最安全的方法——尤其是在你不熟悉 Linux 內核的情況下。問題是：發行商或附加存儲庫提供的大多數版本都是從修改過的 Linux 原始碼構建的。因此它們不是普通的，通常不適合於測試和問題報告：這些更改可能會導致您面臨的問題或以某種方式影響問題。

但是如果您使用的是流行的 Linux 發行版，那麼您就很幸運了：對於大部分的發行版，您可以在網上找到包含最新主線或穩定版本 Linux 內核包的存儲庫。使用這些是完全可以的，只要從存儲庫的描述中確認它們是普通的或者至少接近普通。此外，請確保軟體包包含 [kernel.org](#) 上提供的最新版本內核。如果這些軟體包的時間超過一周，那麼它們可能就不合適了，因為新的主線和穩定版內核通常至少每周發布一次。

請注意，您以後可能需要手動構建自己的內核：有時這是調試或測試修復程序所必需的，如後文所述。還要注意，預編譯的內核可能缺少在出現 `panic`、`Oops`、`warning` 或 `BUG` 時解碼內核列印的消息所需的調試符號；如果您計劃解碼這些消息，最好自己編譯內核（有關詳細信息，請參閱本小節結尾和「解碼失敗信息」小節）。

**使用 git：**熟悉 `git` 的開發者和有經驗的 Linux 用戶通常最好直接從 [kernel.org 上的官方開發倉庫](#) 中獲取最新的 Linux 內核原始碼。這些很可能比最新的主線預發布版本更新一些。不用擔心：它們和正式的預發布版本一樣可靠，除非內核的開發周期目前正處於合併窗口中。不過即便如此，它們也是相當可靠的。

**常規方法：**不熟悉 `git` 的人通常最好從 [kernel.org](#) 下載源碼的 tar 存檔包。

如何實際構建一個內核並不在這裡描述，因為許多網站已經解釋了必要的步驟。如果你是新手，可以考慮按照那些建議使用 `make localmodconfig` 來做，它將嘗試獲取你當前內核的配置，然後根據你的系統進行一些調整。這樣做並不能使編譯出來的內核更好，但可以更快地編譯。

注意：如果您正在處理來自內核的 `panic`、`Oops`、`warning` 或 `BUG`，請在配置內核時嘗試啓用 `CONFIG_KALLSYMS` 選項。此外，還可以啓用 `CONFIG_DEBUG_KERNEL` 和 `CONFIG_DEBUG_INFO`；後者是相關選項，但只有啓用前者才能開啓。請注意，`CONFIG_DEBUG_INFO` 會需要更多儲存空間來構建內核。但這是值得的，因為這些選項將允許您稍後精確定位觸發問題的確切代碼行。下面的「解碼失敗信息」一節對此進行了更詳細的解釋。

但請記住：始終記錄遇到的問題，以防難以重現。發送未解碼的報告總比不報告要好。

### 檢查「汙染」標誌

確保您剛剛安裝的內核在運行時不會「汙染」自己。

正如上面已經詳細介紹過的：當發生一些可能會導致一些看起來完全不相關的後續錯誤的事情時，內核會設置一個「汙染」標誌。這就是為什麼你需要檢查你剛剛安裝的內核是否有設置此標誌。如果有的話，幾乎在任何情況下你都需要在報告問題之前先消除它。詳細的操作方法請看上面的章節。

### 用新內核重現問題

在您剛剛安裝的內核中復現這個問題。如果它沒有出現，請查看下方只發生在穩定版和長期支持內核的問題的說明。

檢查這個問題是否發生在你剛剛安裝的新 Linux 內核版本上。如果新內核已經修復了，可以考慮使用此版本線，放棄報告問題。但是請記住，只要它沒有在 [kernel.org](#) 的穩定版和長期版（以及由這些版本衍生出來的廠商內核）中得到修復，其他用戶可能仍然會受到它的困擾。如果你喜歡使用其中的一個，或者只是想幫助它們的用戶，請前往下面的「報告只發生在較舊內核版本線的問題」一節。

### 優化復現問題的描述

優化你的筆記：試著找到並寫出最直接的復現問題的方法。確保最終結果包含所有重要的細節，同時讓第一次聽說的人容易閱讀和理解。如果您在此過程中學到了一些東西，請考慮再次搜索關於該問題的現有報告。

過於複雜的報告會讓別人很難理解。因此請儘量找到一個可以直接描述、易於以書面形式理解的再現方法。包含所有重要的細節，但同時也要儘量保持簡短。

在這在前面的步驟中，你很可能已經了解了一些關於你所面臨的問題的點。利用這些知識，再次搜索可以轉而加入的現有報告。

### 解碼失敗信息

如果失敗涉及「panic」、「Oops」、「warning」或「BUG」，請考慮解碼內核日誌以查找觸發錯誤的代碼行。

當內核檢測到內部問題時，它會記錄一些有關已執行代碼的信息。這使得在原始碼中精確定位觸發問題的行並顯示如何調用它成為可能。但只有在配置內核時啟用了 CONFIG\_DEBUG\_INFO 和 CONFIG\_KALLSYMS 選項時，這種方法才起效。如果已啓用此選項，請考慮解碼內核日誌中的信息。這將使我們更容易理解是什麼導致了「panic」、「Oops」、「warning」或「BUG」，從而增加了有人提供修復的機率。

解碼可以通過 Linux 原始碼樹中的腳本來完成。如果您運行的內核是之前自己編譯的，這樣這樣調用它：

```
[user@something ~]$ sudo dmesg | ./linux-5.10.5/scripts/decode_stacktrace.sh ./linux-5.10.5/
↳ vmlinux
/usr/lib/debug/lib/modules/5.10.10-4.1.x86_64/vmlinux /usr/src/kernels/5.10.10-4.1.x86_64/
```

如果您運行的是打包好的普通內核，則可能需要安裝帶有調試符號的相應包。然後按以下方式調用腳本（如果發行版未打包，則可能需要從 Linux 原始碼獲取）：

```
[user@something ~]$ sudo dmesg | ./linux-5.10.5/scripts/decode_stacktrace.sh \
/usr/lib/debug/lib/modules/5.10.10-4.1.x86_64/vmlinux /usr/src/kernels/5.10.10-4.1.x86_64/
```

腳本將解碼如下的日誌行，這些日誌行顯示內核在發生錯誤時正在執行的代碼的地址：

```
[ 68.387301] RIP: 0010:test_module_init+0x5/0xffa [test_module]
```

解碼之後，這些行將變成這樣：

```
[ 68.387301] RIP: 0010:test_module_init (/home/username/linux-5.10.5/test-module/test-module.
↪c:16) test_module
```

在本例中，執行的代碼是從文件「~/linux-5.10.5/test-module/test-module.c」構建的，錯誤出現在第 16 行的指令中。

該腳本也會如此解碼以「Call trace」開頭的部分中提到的地址，該部分顯示出現問題的函數的路徑。此外，腳本還會顯示內核正在執行的代碼部分的彙編輸出。

注意，如果你沒法做到這一點，只需跳過這一步，並在報告中說明原因。如果你幸運的話，可能無需解碼。如果需要的話，也許有人會幫你做這件事情。還要注意，這只是解碼內核堆棧跟蹤的幾種方法之一。有時需要採取不同的步驟來檢索相關的詳細信息。別擔心，如果您碰到的情況需要這樣做，開發人員會告訴您該怎麼做。

## 對回歸的特別關照

如果您的問題是回歸問題，請儘可能縮小引入問題時的範圍。

Linux 首席開發者 Linus Torvalds 認為 Linux 內核永遠不應惡化，這就是為什麼他認為回歸是不可接受的，並希望看到它們被迅速修復。這就是為什麼引入了回歸的改動導致的問題若無法通過其他方式快速解決，通常會被迅速撤銷。因此，報告回歸有點像「王炸」，會迅速得到修復。但要做到這一點，需要知道導致回歸的變化。通常情況下，要由報告者來追查罪魁禍首，因為維護者往往沒有時間或手頭設置不便來自行重現它。

有一個叫做「二分」的過程可以來尋找變化，這在「[二分 \(bisect\) 缺陷](#)」文檔中進行了詳細的描述，這個過程通常需要你構建十到二十個內核鏡像，每次都嘗試在構建下一個鏡像之前重現問題。是的，這需要花費一些時間，但不用擔心，它比大多數人想像的要快得多。多虧了「binary search 二進位搜索」，這將引導你在原始碼管理系統中找到導致回歸的提交。一旦你找到它，就在網上搜索其主題、提交 ID 和縮短的提交 ID (提交 ID 的前 12 個字符)。如果有的話，這將引導您找到關於它的現有報告。

需要注意的是，二分法需要一點竅門，不是每個人都懂得訣竅，也需要相當多的努力，不是每個人都願意投入。儘管如此，還是強烈建議自己進行一次二分。如果你真的不能或者不想走這條路，至少要找出是哪個主線內核引入的回歸。比如說從 5.5.15 切換到 5.8.4 的時候出現了一些問題，那麼至少可以嘗試一下相近的所有主線版本（5.6、5.7 和 5.8）來檢查它是什麼時候出現的。除非你想在一個穩定版或長期支持內核中找到一個回歸，否則要避免測試那些編號有三段的版本（5.6.12、5.7.8），因為那會使結果難以解釋，可能會讓你的測試變得無用。一旦你找到了引入回歸的主要版本，就可以放心地繼續報告了。但請記住：在不知道

罪魁禍首的情況下，開發人員是否能夠提供幫助取決於手頭的問題。有時他們可能會從報告中確認是什麼出現了問題，並能修復它；有時他們可能無法提供幫助，除非你進行二分。

當處理回歸問題時，請確保你所面臨的問題真的是由內核引起的，而不是由其他東西引起的，如上文所述。

在整個過程中，請記住：只有當舊內核和新內核的配置相似時，問題才算回歸。最好的方法是：把配置文件 (`.config`) 從舊的工作內核直接複製到你嘗試的每個新內核版本。之後運行 `make oldnoconfig` 來調整它以適應新版本的需要，而不啓用任何新的功能，因為那些功能也可能導致回歸。

### 撰寫並發送報告

通過詳細描述問題來開始編寫報告。記得包括以下條目：您為復現而安裝的最新內核版本、使用的 Linux 發行版以及關於如何復現該問題的說明。如果可能，將內核構建配置 (`.config`) 和 ```dmesg``` 的輸出放在網上的某個地方，並連結到它。包含或上傳所有其他可能相關的信息，如 `Oops` 的輸出/截圖或來自 ```lspci``` 的輸出。一旦你寫完了這個主要部分，請在上方插入一個正常長度的段落快速概述問題和影響。再在此之上添加一個簡單描述問題的句子，以得到人們的閱讀。現在給出一個更短的描述性標題或主題。然後就可以像 `MAINTAINERS` 文件告訴你的那樣發送或提交報告了，除非你在處理一個「高優先級問題」：它們需要按照下面「高優先級問題的特殊處理」所述特別關照。

現在你已經準備好了一切，是時候寫你的報告了。上文前言中連結的三篇文檔對如何寫報告做了部分解釋。這就是為什麼本文將只提到一些基本的內容以及 Linux 內核特有的東西。

有一點是符合這兩類的：你的報告中最關鍵的部分是標題/主題、第一句話和第一段。開發者經常會收到許多郵件。因此，他們往往只是花幾秒鐘的時間瀏覽一下郵件，然後再決定繼續下一封或仔細查看。因此，你報告的開頭越好，有人研究並幫助你的機會就越大。這就是為什麼你應該暫時忽略他們，先寫出詳細的報告。  
;-)

### 每份報告都應提及的事項

詳細描述你的問題是如何發生在你安裝的新純淨內核上的。試著包含你之前寫的和優化過的分步說明，概述你和其他人如何重現這個問題；在極少數無法重現的情況下，儘量描述你做了什麼來觸發它。

還應包括其他人為了解該問題及其環境而可能需要的所有相關信息。實際需要的東西在很大程度上取決於具體問題，但有些事項你總是應該包括在內：

- `cat /proc/version` 的輸出，其中包含 Linux 內核版本號和構建時的編譯器。
- 機器正在運行的 Linux 發行版 (`hostnamectl | grep 'Operating System'`)
- CPU 和作業系統的架構 (`uname -mi`)
- 如果您正在處理回歸，並進行了二分，請提及導致回歸的變更的主題和提交 ID。

許多情況下，讓讀你報告的人多了解兩件事也是明智之舉：

- 用於構建 Linux 內核的配置（「`.config`」文件）

- 內核的信息，你從 `dmesg` 得到的信息寫到一個文件里。確保它以像「Linux version 5.8-1 ([foo-bar@example.com](mailto:foo-bar@example.com)) (gcc (GCC) 10.2.1, GNU ld version 2.34) #1 SMP Mon Aug 3 14:54:37 UTC 2020」這樣的行開始，如果沒有，那麼第一次啓動階段的重要信息已經被丟棄了。在這種情況下，可以考慮使用 `journalctl -b 0 -k`；或者你也可以重啓，重現這個問題，然後調用 `dmesg`。

這兩個文件很大，所以直接把它們放到你的報告中是個壞主意。如果你是在缺陷跟蹤器中提交問題，那麼將它們附加到工單中。如果你通過郵件報告問題，不要用附件附上它們，因為那會使郵件變得太大的，可以按下列之一做：

- 將文件上傳到某個公開的地方（你的網站，公共文件粘貼服務，在 [bugzilla.kernel.org](https://bugzilla.kernel.org) 上創建的工單……），並在你的報告中放上連結。理想情況下請使用允許這些文件保存很多年的地方，因為它們可能在很多年後對別人有用；例如 5 年或 10 年後，一個開發者正在修改一些代碼，而這些代碼正是為了修復你的問題。
- 把文件放在一邊，然後說明你會在他人回復時再單獨發送。只要記得報告發出去後，真正做到這一點就可以了。;-)

## 提供這些東西可能是明智的

根據問題的不同，你可能需要提供更多的背景數據。這裡有一些關於提供什麼比較好的建議：

- 如果你處理的是內核的「warning」、「OOPS」或「panic」，請包含它。如果你不能複製粘貼它，試著用 `netconsole` 網絡終端遠程跟蹤或者至少拍一張屏幕的照片。
- 如果問題可能與你的電腦硬體有關，請說明你使用的是什麼系統。例如，如果你的顯卡有問題，請提及它的製造商，顯卡的型號，以及使用的晶片。如果是筆記本電腦，請提及它的型虧名稱，但儘量確保意義明確。例如「戴爾 XPS 13」就不很明確，因為它可能是 2012 年的那款，那款除了看起來和現在銷售的沒有什麼不同之外，兩者沒有任何共同之處。因此，在這種情況下，要加上準確的型號，例如 2019 年內推出的 XPS 13 型號為「9380」或「7390」。像「聯想 Thinkpad T590」這樣的名字也有些含糊不清：這款筆記本有帶獨立顯卡和不帶的子型號，所以要儘量找到準確的型虧名稱或註明主要部件。
- 說明正在使用的相關軟體。如果你在加載模塊時遇到了問題，你要說明正在使用的 `kmod`、`systemd` 和 `udev` 的版本。如果其中一個 DRM 驅動出現問題，你要說明 `libdrm` 和 `Mesa` 的版本；還要說明你的 `Wayland` 合成器或 `X-Server` 及其驅動。如果你有文件系統問題，請註明相應的文件系統實用程序的版本（`e2fsprogs`, `btrfs-progs`, `xfsprogs`……）。
- 從內核中收集可能有用的額外信息。例如，`lspci -nn` 的輸出可以幫助別人識別你使用的硬體。如果你的硬體有問題，你甚至可以給出 `sudo lspci -vvv` 的結果，因為它提供了組件是如何配置的信息。對於一些問題，可能最好包含 `/proc/cpuinfo`，`/proc/ioports`，`/proc/iomem`，`/proc/modules` 或 `/proc/scsi/scsi` 等文件的內容。一些子系統還提供了收集相關信息的工具。`alsa-info.sh` 就是這樣一個工具，它是音頻/聲音子系統開發者提供的。

這些例子應該會給你一些知識點，讓你知道附上什麼數據可能是明智的，但你自己也要想一想，哪些數據對別人會有幫助。不要太擔心忘記一些東西，因為開發人員會要求提供他們需要的額外細節。但從一開始就把所有重要的東西都提供出來，會增加別人仔細查看的機會。

### 重要部分：報告的開頭

現在你已經準備好了報告的詳細部分，讓我們進入最重要的部分：開頭幾句。現在到報告的最前面，在你剛才寫的部分之前加上類似「The detailed description:」（詳細描述）這樣的內容，並在最前面插入兩個新行。現在寫一個正常長度的段落，大致概述這個問題。去掉所有枯燥的細節，把重點放在讀者需要知道的關鍵部分，以讓人了解這是怎麼回事；如果你認為這個缺陷影響了很多用戶，就提一下這點來吸引大家關注。

做好這一點後，在頂部再插入兩行，寫一句話的摘要，快速解釋報告的內容。之後你要更加抽象，為報告寫一個更短的主題/標題。

現在你已經寫好了這部分，請花點時間來優化它，因為它是你的報告中最重要的部分：很多人會先讀這部分，然後才會決定是否值得花時間閱讀其他部分。

現在就像 MAINTAINERS 維護者文件告訴你的那樣發送或提交報告，除非它是前面概述的那些「高優先級問題」之一：在這種情況下，請先閱讀下一小節，然後再發送報告。

### 高優先級問題的特殊處理

高優先級問題的報告需要特殊處理。

**非常嚴重的缺陷**：確保在主題或工單標題以及第一段中明顯標出 severeness （非常嚴重的）。

**回歸**：如果問題是一個回歸，請在郵件的主題或缺陷跟蹤器的標題中添加 [REGRESSION]。如果您沒有進行二分，請至少註明您測試的最新主線版本（比如 5.7）和出現問題的最新版本（比如 5.8）。如果您成功地進行了二分，請註明導致回歸的提交 ID 和主題。也請添加該變更的作者到你的報告中；如果您需要將您的缺陷提交到缺陷跟蹤器中，請將報告以私人郵件的形式轉發給他，並註明報告提交地點。

**安全問題**：對於這種問題，你將必須評估：如果細節被公開披露，是否會對其他用戶產生短期風險。如果不會，只需按照所述繼續報告問題。如果有此風險，你需要稍微調整一下報告流程。

- 如果 MAINTAINERS 文件指示您通過郵件報告問題，請不要抄送任何公共郵件列表。
- 如果你應該在缺陷跟蹤器中提交問題，請確保將工單標記為「私有」或「安全問題」。如果缺陷跟蹤器沒有提供保持報告私密性的方法，那就別想了，把你的報告以私人郵件的形式發送給維護者吧。

在這兩種情況下，都一定要將報告發到 MAINTAINERS 文件中「安全聯絡」部分列出的地址。理想的情況是在發送報告的時候直接抄送他們。如果您在缺陷跟蹤器中提交了報告，請將報告的文本轉發到這些地址；但請在報告的頂部加上注釋，表明您提交了報告，並附上工單連結。

更多信息請參見「[安全缺陷](#)」。

## 發布報告後的責任

等待別人的反應，繼續推進事情，直到你能夠接受這樣或那樣的結果。因此，請公開和及時地回應任何詢問。測試提出的修復。積極地測試：至少重新測試每個新主線版本的首個候選版本（*RC*），並報告你的結果。如果出現拖延，就友好地提醒一下。如果你沒有得到任何幫助或者未能滿意，請試著自己幫助自己。

如果你的報告非常優秀，而且你真的很幸運，那麼某個開發者可能會立即發現導致問題的原因；然後他們可能會寫一個補丁來修復、測試它，並直接發送給主線集成，同時標記它以便以後回溯到需要它的穩定版和長期支持內核。那麼你需要做的就是回復一句「Thank you very much」（非常感謝），然後在發布後換上修復好的版本。

但這種理想狀況很少發生。這就是為什麼你把報告拿出來之後工作才開始。你要做的事情要視情況而定，但通常會是下面列出的事情。但在深入研究細節之前，這裡有幾件重要的事情，你需要記住這部分的過程。

## 關於進一步互動的一般建議

**總是公開回復：**當你在缺陷跟蹤器中提交問題時，一定要在那裡回復，不要私下聯繫任何開發者。對於郵件報告，在回復您收到的任何郵件時，總是使用「全部回復」功能。這包括帶有任何你可能想要添加到你的報告中的額外數據的郵件：進入郵件應用程序「已發送」文件夾，並在郵件上使用「全部回復」來回復報告。這種方法可以確保公共郵件列表和其他所有參與者都能及時了解情況；它還能保持郵件線程的完整性，這對於郵件列表將所有相關郵件歸為一類是非常重要的。

只有兩種情況不適合在缺陷跟蹤器或「全部回復」中發表評論：

- 有人讓你私下發東西。
- 你被告知要發送一些東西，但注意到其中包含需要保密的敏感信息。在這種情況下，可以私下發送給要求發送的開發者。但要在工單或郵件中註明你是這麼做的，這樣其他人就知道你尊重了這個要求。

**在請求解釋或幫助之前先研究一下：**在這部分過程中，有人可能會告訴你用尚未掌握的技能做一些事情。例如你可能會被要求使用一些你從未聽說過的測試工具；或者你可能會被要求在 Linux 內核原始碼上應用一個補丁來測試它是否有幫助。在某些情況下，發個回復詢問如何做就可以了。但在走這條路之前，儘量通過在網際網路上搜索自行找到答案；或者考慮在其他地方詢問建議。比如詢問朋友，或者到你平時常去的聊天室或論壇發帖諮詢。

**要有耐心：**如果你真的很幸運，你可能會在幾個小時內收到對你的報告的答覆。但大多數情況下會花費更多的時間，因為維護者分散在全球各地，因此可能在不同的時區——在那裡他們已經享受著遠離鍵盤的夜晚。

一般來說，內核開發者需要一到五個工作日來回復報告。有時會花費更長的時間，因為他們可能正忙於合併窗口、其他工作、參加開發者會議，或者只是在享受一個漫長的暑假。

「高優先級的問題」（見上面的解釋）例外：維護者應該儘快解決這些問題；這就是為什麼你應該最多等待一個星期（如果是緊急的事情，則只需兩天），然後再發送友好的提醒。

有時維護者可能沒有及時回復；有時候可能會出現分歧，例如一個問題是否符合回歸的條件。在這種情況下，在郵件列表上提出你的顧慮，並請求其他人公開或私下回復如何繼續推進。如果失敗了，可能應該讓更高級

別的維護者介入。如果是 WiFi 驅動，那就是無線維護者；如果沒有更高級別的維護者，或者其他一切努力都失敗了，那這可能是一種罕見的、可以讓 Linus Torvalds 參與進來的情況。

**主動測試**：每當一個新的主線內核版本的第一個預發布版本（rc1）發布的時候，去檢查一下這個問題是否得到了解決，或者是否有什麼重要的變化。在工單中或在回復報告的郵件中提及結果（確保所有參與討論的人都被抄送）。這將表明你的承諾和你願意幫忙。如果問題持續存在，它也會提醒開發者確保他們不會忘記它。其他一些不定期的重新測試（例如用 rc3、rc5 和最終版本）也是一個好主意，但只有在相關的東西發生變化或者你正在寫什麼東西的時候才報告你的結果。

這些些常規的事情就不說了，我們來談談報告後如何幫助解決問題的細節。

### 查詢和測試請求

如果你的報告得到了回復則需履行以下責任：

**檢查與你打交道的人**：大多數情況下，會是維護者或特定代碼區域的開發人員對你的報告做出回應。但由於問題通常是公開報告的，所以回復的可能是任何人——包括那些想要幫忙的人，但最後可能會用他們的問題或請求引導你完全偏離軌道。這很少發生，但這是快速上網搜搜看你正在與誰互動是明智之舉的許多原因之一。通過這樣做，你也可以知道你的報告是否被正確的人聽到，因為如果討論沒有導致滿意的問題解決方案而淡出，之後可能需要提醒維護者（見下文）。

**查詢數據**：通常你會被要求測試一些東西或提供更多細節。儘快提供所要求的信息，因為你已經得到了可能會幫助你的人的注意，你等待的時間越長就有越可能失去關注；如果你不在數個工作日內提供信息，甚至可能出現這種結果。

**測試請求**：當你被要求測試一個診斷補丁或可能的修復時，也要儘量及時測試。但要做得恰當，一定不要急於求成：混淆事情很容易發生，這會給所有人帶來許多困惑。例如一個常見的錯誤是以爲應用了一個帶修復的建議補丁，但事實上並沒有。即使是有經驗的測試人員也會偶爾發生這樣的事情，但當有修復的內核和沒有修復的內核表現得一樣時，他們大多時候會注意到。

### 當沒有任何實質性進展時該怎麼辦

有些報告不會得到負有相關責任的 Linux 內核開發者的任何反應；或者圍繞這個問題的討論有所發展，但漸漸淡出，沒有任何實質內容產出。

在這種情況下，要等兩個星期（最好是三個星期）後再發出友好的提醒：也許當你的報告到達時，維護者剛剛離開鍵盤一段時間，或者有更重要的事情要處理。在寫提醒信的時候，要善意地問一下，是否還需要你這邊提供什麼來讓事情推進下去。如果報告是通過郵件發出來的，那就在郵件的第一行回覆你的初始郵件（見上文），其中包括下方的原始報告的完整引用：這是少數幾種情況下，這樣的「TOFU」（Text Over, Fullquote Under 文字在上，完整引用在下）是正確的做法，因為這樣所有的收件人都會以適當的順序立即讓細節到手頭上來。

在提醒之後，再等三周的回覆。如果你仍然沒有得到適當的反饋，你首先應該重新考慮你的方法。你是否可能嘗試接觸了錯誤的人？是不是報告也許令人反感或者太混亂，以至於人們決定完全遠離它？排除這些因素的最好方法是：把報告給一兩個熟悉 FLOSS 問題報告的人看，詢問他們的意見。同時徵求他們關於如何繼

續推進的建議。這可能意味著：準備一份更好的報告，讓這些人在你發出去之前對它進行審查。這樣的方法完全可以；只需說明這是關於這個問題的第二份改進的報告，並附上第一份報告的連結。

如果報告是恰當的，你可以發送第二封提醒信；在其中詢問為什麼報告沒有得到任何回復。第二封提醒郵件的好時機是在新 Linux 內核版本的首個預發布版本（‘rc1’）發布後不久，因為無論如何你都應該在那個時候重新測試並提供狀態更新（見上文）。

如果第二次提醒的結果又在一週內沒有任何反應，可以嘗試聯繫上級維護者詢問意見：即使再忙的維護者在這時候也至少應該發過某種確認。

記住要做好失望的準備：理想狀況下維護者最好對每一個問題報告做出回應，但他們只有義務解決之前列出的「高優先級問題」。所以，如果你得到的回覆是「謝謝你的報告，我目前有更重要的問題要處理，在可預見的未來沒有時間去研究這個問題」，那請不要太沮喪。

也有可能在缺陷跟蹤器或列表中進行了一些討論之後，什麼都沒有發生，提醒也無助於激勵大家進行修復。這種情況可能是毀滅性的，但在 Linux 內核開發中確實會發生。這些和其他得不到幫助的原因在本文結尾處的「為什麼有些問題在被報告後沒有得到任何回應或者仍然沒有修復」中進行了解釋。

如果你沒有得到任何幫助或問題最終沒有得到解決，不要沮喪：Linux 內核是 FLOSS，因此你仍然可以自己幫助自己。例如，你可以試著找到其他受影響的人，和他們一起合作來解決這個問題。這樣的團隊可以一起準備一份新的報告，提到團隊有多少人，為什麼你們認為這是應該得到解決的事情。也許你們還可以一起縮小確切原因或引入回歸的變化，這往往會使修復更容易。而且如果運氣好的話，團隊中可能會有懂點編程的人，也許能寫出一個修複方案。

## 「報告穩定版和長期支持內核線的回歸」的參考

本小節提供了在穩定版和長期支持內核線中面對回歸時需要執行的步驟的詳細信息。

### 確保特定版本線仍然受支持

檢查內核開發人員是否仍然維護你關心的 Linux 內核版本線：去 [kernel.org](http://kernel.org) 的首頁，確保此特定版本線的最新版沒有「[EOL]」標記。

大多數內核版本線只支持三個月左右，因為延長維護時間會帶來相當多的工作。因此，每年只會選擇一個版本來支持至少兩年（通常是六年）。這就是為什麼你需要檢查內核開發者是否還支持你關心的版本線。

注意，如果 [kernel.org](http://kernel.org) 在首頁上列出了兩個「穩定」版本，你應該考慮切換到較新的版本，而忘掉較舊的版本：對它的支持可能很快就會結束。然後，它將被標記為「生命周期結束」(EOL)。達到這個程度的版本線仍然會在 [kernel.org](http://kernel.org) 首頁上被顯示一兩周，但不適合用於測試和報告。

### 搜索穩定版郵件列表

檢查 Linux 穩定版郵件列表中的現有報告。

也許你所面臨的問題已經被發現，並且已經或即將被修復。因此，請在 Linux 穗定版郵件列表的檔案 中搜索類似問題的報告。如果你找到任何匹配的問題，可以考慮加入討論，除非修復工作已經完成並計劃很快得到應用。

### 用最新版本復現問題

從特定的版本線安裝最新版本作為純淨內核。確保這個內核沒有被汙染，並且仍然存在問題，因為問題可能已經在那裡被修復了。

在投入更多時間到這個過程中之前，你要檢查這個問題是否在你關注的版本線的最新版本中已經得到了修復。這個內核需要是純淨的，在問題發生之前不應該被汙染，正如上面已經在測試主線的過程中詳細介紹過的一樣。

您是否是第一次注意到供應商內核的回歸？供應商的更改可能會發生變化。你需要重新檢查排除來這個問題。當您從 5.10.4-vendor.42 更新到 5.10.5-vendor.43 時，記錄損壞的信息。然後在測試了前一段中所述的最新 5.10 版本之後，檢查 Linux 5.10.4 的普通版本是否也可以正常工作。如果問題在那裡出現，那就不符合上游回歸的條件，您需要切換回主逐步指南來報告問題。

### 報告回歸

向 Linux 穗定版郵件列表發送一個簡短的問題報告 ([stable@vger.kernel.org](mailto:stable@vger.kernel.org))。大致描述問題，並解釋如何復現。講清楚首個出現問題的版本和最後一個工作正常的版本。然後等待進一步的指示。

當報告在穩定版或長期支持內核線內發生的回歸（例如在從 5.10.4 更新到 5.10.5 時），一份簡短的報告足以快速報告問題。因此只需要粗略的描述。

但是請注意，如果您能夠指明引入問題的確切版本，這將對開發人員有很大幫助。因此如果有時間的話，請嘗試使用普通內核找到該版本。讓我們假設發行版發布 Linux 內核 5.10.5 到 5.10.8 的更新時發生了故障。那麼按照上面的指示，去檢查該版本線中的最新內核，比如 5.10.9。如果問題出現，請嘗試普通 5.10.5，以確保供應商應用的補丁不會干擾。如果問題沒有出現，那麼嘗試 5.10.7，然後直到 5.10.8 或 5.10.6（取決於結果）找到第一個引入問題的版本。在報告中寫明這一點，並指出 5.10.9 仍然存在故障。

前一段基本粗略地概述了「二分」方法。一旦報告出來，您可能會被要求做一個正確的報告，因為它允許精確地定位導致問題的確切更改（然後很容易被恢復以快速修復問題）。因此如果時間允許，考慮立即進行適當的二分。有關如何詳細信息，請參閱「對回歸的特別關照」部分和文檔「二分 (bisect) 缺陷」。

## 「報告僅在舊內核版本線中發生的問題」的參考

本節詳細介紹了如果無法用主線內核重現問題，但希望在舊版本線（又稱穩定版內核和長期支持內核）中修復問題時需要採取的步驟。

### 有些修復太複雜

請做好準備，接下來的幾個步驟可能無法在舊版本中解決問題：修復可能太大或太冒險，無法移植到那裡。

即使是微小的、看似明顯的代碼變化，有時也會帶來新的、完全意想不到的問題。穩定版和長期支持內核的維護者非常清楚這一點，因此他們只對這些內核進行符合「[所有你想知道的事情 - 關於 linux 穩定版發布](#)」中所列出的規則的修改。

複雜或有風險的修改不符合條件，因此只能應用於主線。其他的修復很容易被回溯到最新的穩定版和長期支持內核，但是風險太大，無法集成到舊版內核中。所以要注意你所希望的修復可能是那些不會被回溯到你所關心的版本線的修復之一。在這種情況下，你將別無選擇，要麼忍受這個問題，要麼切換到一個較新的 Linux 版本，除非你想自己把修復補丁應用到你的內核中。

### 通用準備

執行上面「報告僅在舊內核版本線中發生的問題」一節中的前三個步驟。

您需要執行本指南另一節中已經描述的幾個步驟。這些步驟將讓您：

- 檢查內核開發人員是否仍然維護您關心的 Linux 內核版本行。
- 在 Linux 穩定郵件列表中搜索退出的報告。
- 檢查最新版本。

### 檢查代碼歷史和搜索現有的討論

在 Linux 內核版本控制系統中搜索修復主線問題的更改，因為它的提交消息可能會告訴你修復是否已經計劃好了支持。如果你沒有找到，搜索適當的郵件列表，尋找討論此類問題或同行評議可能修復的帖子；然後檢查討論是否認為修復不適合支持。如果支持根本不被考慮，加入最新的討論，詢問是否有可能。

在許多情況下，你所處理的問題會發生在主線上，但已在主線上得到了解決。修正它的提交也需要被回溯才能解決這個問題。這就是為什麼你要搜索它或任何相關討論。

- 首先嘗試在存放 Linux 內核原始碼的 Git 倉庫中找到修復。你可以通過 [kernel.org 上的網頁](#) 或 [GitHub 上的鏡像](#) 來實現；如果你有一個本地克隆，你也可以在命令行用 `git log --grep=<pattern>` 來搜索。

如果你找到了修復，請查看提交消息的尾部是否包含了類似這樣的「穩定版標籤」：

Cc: <[sstable@vger.kernel.org](mailto:sstable@vger.kernel.org)> # 5.4+

像上面這行，開發者標記了安全修復可以回傳到 5.4 及以後的版本。大多數情況下，它會在兩周內被應用到那裡，但有時需要更長的時間。

- 如果提交沒有告訴你任何東西，或者你找不到修復，請再找找關於這個問題的討論。用你最喜歡的搜尋引擎搜索網絡，以及 [Linux kernel developers mailing list](#) 內核開發者郵件列表 的檔案。也可以閱讀上面的 定位導致問題的內核區域一節，然後按照說明找到導致問題的子系統：它的缺陷跟蹤器或郵件列表存檔中可能有你要找的答案。
- 如果你看到了一個計劃的修復，請按上所述在版本控制系統中搜索它，因為提交可能會告訴你是否可以進行回溯。
  - 檢查討論中是否有任何跡象表明，該修復程序可能風險太大，無法追溯到你關心的版本線。如果是這樣的話，你必須忍受這個問題，或者切換到應用了修復的內核版本線。
  - 如果修復的問題未包含穩定版標籤，並且沒有討論過追溯問題，請加入討論：如果合適的話，請提及你所面對的問題的版本，以及你希望看到它被修復。

### 請求建議

前面的步驟之一應該會給出一個解決方案。如果仍未能成功，請向可能引起問題的子系統的維護人員詢問建議；抄送特定子系統的郵件列表以及穩定版郵件列表。

如果前面的三個步驟都沒有讓你更接近解決方案，那麼只剩下一個選擇：請求建議。在你發給可能是問題根源的子系統的維護者的郵件中這樣做；抄送子系統的郵件列表以及穩定版郵件列表 ([sstable@vger.kernel.org](mailto:sstable@vger.kernel.org))。

### 為什麼有些問題在報告後沒有任何回應或仍未解決？

當向 Linux 開發者報告問題時，要注意只有「高優先級的問題」（回歸、安全問題、嚴重問題）才一定會得到解決。如果維護者或其他人都失敗了，Linus Torvalds 他自己會確保這一點。他們和其他內核開發者也會解決很多其他問題。但是要知道，有時他們也會不能或不願幫忙；有時甚至沒有人發報告給他們。

最好的解釋就是那些內核開發者常常是在業餘時間為 Linux 內核做出貢獻。內核中的不少驅動程序都是由這樣的程式設計師編寫的，往往只是因為他們想讓自己的硬體可以在自己喜歡的作業系統上使用。

這些程式設計師大多數時候會很樂意修復別人報告的問題。但是沒有人可以強迫他們這樣做，因為他們是自願貢獻的。

還有一些情況下，這些開發者真的很想解決一個問題，但卻不能解決：有時他們缺乏硬體編程文檔來解決問題。這種情況往往由於公開的文檔太簡陋，或者驅動程序是通過逆向工程編寫的。

業餘開發者遲早也會不再關心某驅動。也許他們的測試硬體壞了，被更高級的玩意取代了，或者是太老了以至於只能在計算機博物館裡找到。有時開發者根本就不關心他們的代碼和 Linux 了，因為在他們的生活中一些不同的東西變得更重要了。在某些情況下，沒有人願意接手維護者的工作——也沒有人可以被強迫，因為對

Linux 內核的貢獻是自願的。然而被遺棄的驅動程序仍然存在於內核中：它們對人們仍然有用，刪除它們可能導致回歸。

對於那些為 Linux 內核工作而獲得報酬的開發者來說，情況並沒有什麼不同。這些人現在貢獻了大部分的變更。但是他們的僱主遲早也會停止關注他們的代碼或者讓程序員專注於其他事情。例如，硬體廠商主要通過銷售新硬體來賺錢；因此，他們中的不少人並沒有投入太多時間和精力來維護他們多年前就停止銷售的東西的 Linux 內核驅動。企業級 Linux 發行商往往持續維護的時間比較長，但在新版本中往往會把對老舊和稀有硬體的支持放在一邊，以限制範圍。一旦公司拋棄了一些代碼，往往由業餘貢獻者接手，但正如上面提到的：他們遲早也會放下代碼。

優先級是一些問題沒有被修復的另一個原因，因為維護者相當多的時候是被迫設置這些優先級的，因為在 Linux 上工作的時間是有限的。對於業餘時間或者僱主給予他們的開發人員用於上游內核維護工作的時間也是如此。有時維護人員也會被報告淹沒，即使一個驅動程序幾乎完美地工作。為了不被完全纏住，程式設計師可能別無選擇，只能對問題報告進行優先級排序而拒絕其中的一些報告。

不過這些都不用太過擔心，很多驅動都有積極的維護者，他們對儘可能多的解決問題相當感興趣。

## 結束語

與其他免費/自由 & 開源軟體 (Free/Libre & Open Source Software, FLOSS) 相比，向 Linux 內核開發者報告問題是很難的：這個文檔的長度和複雜性以及字裡行間的內涵都說明了這一點。但目前就是這樣了。這篇文字的主要作者希望通過記錄現狀來為以後改善這種狀況打下一些基礎。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

**Original** [.../..../admin-guide/security-bugs](https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/admin-guide/security-bugs.rst)

譯者 吳 想 成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)> 胡 皓 文 Hu Haowen  
<[src.res@email.cn](mailto:<src.res@email.cn>)>

### 安全缺陷

Linux 內核開發人員非常重視安全性。因此我們想知道何時發現了安全漏洞，以便儘快修復和披露。請向 Linux 內核安全團隊報告安全漏洞。

### 聯絡

可以通過電子郵件 <[security@kernel.org](mailto:security@kernel.org)> 聯繫 Linux 內核安全團隊。這是一個安全人員的私有列表，他們將幫助驗證錯誤報告並開發和發布修復程序。如果您已經有了一個修復，請將其包含在您的報告中，這樣可以大大加快進程。安全團隊可能會從區域維護人員那裡獲得額外的幫助，以理解和修復安全漏洞。

與任何缺陷一樣，提供的信息越多，診斷和修復就越容易。如果您不清楚哪些信息有用，請查看「[報告問題](#)」中概述的步驟。任何利用漏洞的攻擊代碼都非常有用，未經報告者同意不會對外發布，除非已經公開。

請儘可能發送無附件的純文本電子郵件。如果所有的細節都藏在附件里，那麼就很難對一個複雜的問題進行上下文引用的討論。把它想像成一個[常規的補丁提交](#)（即使你還沒有補丁）：描述問題和影響，列出復現步驟，然後給出一個建議的解決方案，所有這些都是純文本的。

### 披露和限制信息

安全列表不是公開渠道。為此，請參見下面的協作。

一旦開發出了健壯的補丁，發布過程就開始了。對公開的缺陷的修復會立即發布。

儘管我們傾向於在未公開缺陷的修復可用時即發布補丁，但應報告者或受影響方的請求，這可能會被推遲到發布過程開始後的 7 日內，如果根據缺陷的嚴重性需要更多的時間，則可額外延長到 14 天。推遲發布修復的唯一有效原因是為了適應 QA 的邏輯和需要發布協調的大規模部署。

雖然可能與受信任的個人共享受限信息以開發修復，但未經報告者許可，此類信息不會與修復程序一起發布或發布在任何其他披露渠道上。這包括但不限於原始錯誤報告和後續討論（如有）、漏洞、CVE 信息或報告者的身份。

換句話說，我們唯一感興趣的是修復缺陷。提交給安全列表的所有其他資料以及對報告的任何後續討論，即使在解除限制之後，也將永久保密。

### 協調

對敏感缺陷（例如那些可能導致權限提升的缺陷）的修復可能需要與私有郵件列表 <[linux-distros@vs.openwall.org](mailto:linux-distros@vs.openwall.org)> 進行協調，以便分發供應商做好準備，在公開披露上游補丁時發布一個已修復的內核。發行版將需要一些時間來測試建議的補丁，通常會要求至少幾天的限制，而供應商更新發布更傾向於周二至周四。若合適，安全團隊可以協助這種協調，或者報告者可以從一開始就包括 linux 發行版。在這種情況下，請記住在電子郵件主題行前面加上「[vs]」，如 linux 發行版 wiki 中所述：<http://oss-security.openwall.org/wiki/mailing-lists/distros#how-to-use-the-lists>。

## CVE 分配

安全團隊通常不分配 CVE，我們也不需要它們來進行報告或修復，因為這會使過程不必要的複雜化，並可能耽誤缺陷處理。如果報告者希望在公開披露之前分配一個 CVE 編號，他們需要聯繫上述的私有 linux-distros 列表。當在提供補丁之前已有這樣的 CVE 編號時，如報告者願意，最好在提交消息中提及它。

## 保密協議

Linux 內核安全團隊不是一個正式的機構實體，因此無法簽訂任何保密協議。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

**Original** [.../.../admin-guide/bug-hunting](https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/admin-guide/bug-hunting)

譯者 吳 想 成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)> 胡 皓 文 Hu Haowen  
<[src.res@email.cn](mailto:<src.res@email.cn>)>

## 追蹤缺陷

內核錯誤報告通常附帶如下堆棧轉儲：

```
-----[ cut here ]-----
WARNING: CPU: 1 PID: 28102 at kernel/module.c:1108 module_put+0x57/0x70
Modules linked in: dvb_usb_gp8psk(-) dvb_usb dvb_core nvidia_drm(P0) nvidia_modeset(P0) snd_
↳hda_codec_hdmi snd_hda_intel snd_hda_codec snd_hwdep snd_hda_core snd_pcm snd_timer snd_
↳soundcore nvidia(P0) [last unloaded: rc_core]
CPU: 1 PID: 28102 Comm: rmmmod Tainted: P WC 0 4.8.4-build.1 #1
Hardware name: MSI MS-7309/MS-7309, BIOS V1.12 02/23/2009
00000000 c12ba080 00000000 00000000 c103ed6a c1616014 00000001 00006dc6
c1615862 00000454 c109e8a7 c109e8a7 00000009 ffffffff 00000000 f13f6a10
f5f5a600 c103ee33 00000009 00000000 00000000 c109e8a7 f80ca4d0 c109f617
Call Trace:
[<c12ba080>] ? dump_stack+0x44/0x64
[<c103ed6a>] ? __warn+0xfa/0x120
[<c109e8a7>] ? module_put+0x57/0x70
[<c109e8a7>] ? module_put+0x57/0x70
[<c103ee33>] ? warn_slowpath_null+0x23/0x30
[<c109e8a7>] ? module_put+0x57/0x70
```

```
[<f80ca4d0>] ? gp8psk_fe_set_frontend+0x460/0x460 [dvb_usb_gp8psk]
[<c109f617>] ? symbol_put_addr+0x27/0x50
[<f80bc9ca>] ? dvb_usb_adapter_frontend_exit+0x3a/0x70 [dvb_usb]
[<f80bb3bf>] ? dvb_usb_exit+0x2f/0xd0 [dvb_usb]
[<c13d03bc>] ? usb_disable_endpoint+0x7c/0xb0
[<f80bb48a>] ? dvb_usb_device_exit+0x2a/0x50 [dvb_usb]
[<c13d2882>] ? usb_unbind_interface+0x62/0x250
[<c136b514>] ? __pm_runtime_idle+0x44/0x70
[<c13620d8>] ? __device_release_driver+0x78/0x120
[<c1362907>] ? driver_detach+0x87/0x90
[<c1361c48>] ? bus_remove_driver+0x38/0x90
[<c13d1c18>] ? usb_deregister+0x58/0xb0
[<c109fb0>] ? SyS_delete_module+0x130/0x1f0
[<c1055654>] ? task_work_run+0x64/0x80
[<c1000fa5>] ? exit_to_usermode_loop+0x85/0x90
[<c10013f0>] ? do_fast_syscall_32+0x80/0x130
[<c1549f43>] ? sysenter_past_esp+0x40/0x6a
---[ end trace 6ebc60ef3981792f ]---
```

這樣的堆棧跟蹤提供了足夠的信息來識別內核原始碼中發生錯誤的那一行。根據問題的嚴重性，它還可能包含「**Oops**」一詞，比如：

```
BUG: unable to handle kernel NULL pointer dereference at (null)
IP: [<c06969d4>] iret_exc+0x7d0/0xa59
*pdp = 000000002258a001 *pde = 0000000000000000
Oops: 0002 [#1] PREEMPT SMP
...
...
```

儘管有 **Oops** 或其他類型的堆棧跟蹤，但通常需要找到出問題的行來識別和處理缺陷。在本章中，我們將參考「**Oops**」來了解需要分析的各種堆棧跟蹤。

如果內核是用 `CONFIG_DEBUG_INFO` 編譯的，那麼可以使用文件：`scripts/decode_stacktrace.sh`。

## 連結的模塊

受到汙染或正在加載/卸載的模塊用「(... )」標記，汙染標誌在 `Documentation/admin-guide/tainted-kernels.rst` 文件中進行了描述，「正在被加載」用「+」標註，「正在被卸載」用「-」標註。

## Oops 消息在哪？

通常，Oops 文本由 `klogd` 從內核緩衝區讀取，然後交給 `syslogd`，後者將其寫入 `syslog` 文件，通常是 `/var/log/messages`（取決於 `/etc/syslog.conf`）。在使用 `systemd` 的系統上，它也可以由 `journald` 守護進程存儲，並通過運行 `journalctl` 命令進行訪問。

有時 `klogd` 會掛掉，這種情況下您可以運行 `dmesg > file` 從內核緩衝區讀取數據並保存它。或者您可以 `cat /proc/kmsg > file`，但是您必須適時中斷以停止傳輸，因為 `kmsg` 是一個「永無止境的文件」。

如果機器嚴重崩潰，無法輸入命令或磁碟不可用，那還有三個選項：

- (1) 手動複製屏幕上的文本，並在機器重新啓動後輸入。很難受，但這是突然崩潰下唯一的選擇。或者你可以用數位相機拍下屏幕——雖然不那麼好，但總比什麼都沒有好。如果消息滾動超出控制台頂部，使用更高解析度（例如 `vga=791`）引導啓動將允許您閱讀更多文本。（警告：這需要 `vesafb`，因此對「早期」的 Oppses 沒有幫助）
- (2) 從串口終端啓動（參見 `Documentation/admin-guide/serial-console.rst`），在另一台機器上運行數據機然後用你喜歡的通信程序捕獲輸出。`Minicom` 運行良好。
- (3) 使用 `Kdump`（參閱 `Documentation/admin-guide/kdump/kdump.rst`），使用 `Documentation/admin-guide/kdump/gdbmacros.txt` 中的 `dmesg gdbmacro` 從舊內存中提取內核環形緩衝區。

## 找到缺陷位置

如果你能指出缺陷在內核原始碼中的位置，則報告缺陷的效果會非常好。這有兩種方法。通常來說使用 `gdb` 會比較容易，不過內核需要用調試信息來預編譯。

### **gdb**

GNU 調試器（GNU debugger，`gdb`）是從 `vmlinux` 文件中找出 OOPS 的確切文件和行號的最佳方法。

在使用 `CONFIG_DEBUG_INFO` 編譯的內核上使用 `gdb` 效果最好。可通過運行以下命令進行設置：

```
$ ./scripts/config -d COMPILE_TEST -e DEBUG_KERNEL -e DEBUG_INFO
```

在用 `CONFIG_DEBUG_INFO` 編譯的內核上，你可以直接從 OOPS 複製 EIP 值：

```
EIP: 0060:[<c021e50e>] Not tainted VLI
```

並使用 GDB 來將其翻譯成可讀形式：

```
$ gdb vmlinux
(gdb) l *0xc021e50e
```

如果沒有啓用 `CONFIG_DEBUG_INFO`，則使用 OOPS 的函數偏移：

```
EIP is at vt_ioctl+0xda8/0x1482
```

並在啓用 `CONFIG_DEBUG_INFO` 的情況下重新編譯內核：

```
$ ./scripts/config -d COMPILE_TEST -e DEBUG_KERNEL -e DEBUG_INFO
$ make vmlinux
$ gdb vmlinux
(gdb) l *vt_ioctl+0xda8
0x1888 is in vt_ioctl (drivers/tty/vt/vt_ioctl.c:293).
```

```
288 {  
289     struct vc_data *vc = NULL;  
290     int ret = 0;  
291  
292     console_lock();  
293     if (VT_BUSY(vc_num))  
294         ret = -EBUSY;  
295     else if (vc_num)  
296         vc = vc_deallocate(vc_num);  
297     console_unlock();
```

或者若您想要更詳細的顯示:

```
(gdb) p vt_ioctl  
$1 = {int (struct tty_struct *, unsigned int, unsigned long)} 0xae0 <vt_ioctl>  
(gdb) l *0xae0+0xda8
```

您也可以使用對象文件作為替代:

```
$ make drivers/tty/  
$ gdb drivers/tty/vt/vt_ioctl.o  
(gdb) l *vt_ioctl+0xda8
```

如果你有調用跟蹤，類似:

```
Call Trace:  
[<ffffffff8802c8e9>] :jbd:log_wait_commit+0xa3/0xf5  
[<ffffffff810482d9>] autoremove_wake_function+0x0/0x2e  
[<ffffffff8802770b>] :jbd:journal_stop+0x1be/0x1ee  
...
```

這表明問題可能在:jbd: 模塊中。您可以在 gdb 中加載該模塊並列出相關代碼:

```
$ gdb fs/jbd/jbd.ko  
(gdb) l *log_wait_commit+0xa3
```

**Note:** 您還可以對堆棧跟蹤處的任何函數調用執行相同的操作，例如:

```
[<f80bc9ca>] ? dvb_usb_adapter_frontend_exit+0x3a/0x70 [dvb_usb]
```

上述調用發生的位置可以通過以下方式看到:

```
$ gdb drivers/media/usb/dvb-usb/dvb-usb.o  
(gdb) l *dvb_usb_adapter_frontend_exit+0x3a
```

## objdump

要調試內核，請使用 `objdump` 並從崩潰輸出中查找十六進位偏移，以找到有效的代碼/匯編行。如果沒有調試符號，您將看到所示例程的彙編程序代碼，但是如果內核有調試符號，C 代碼也將可見（調試符號可以在內核配置菜單的 `hacking` 項中啓用）。例如：

```
$ objdump -r -S -l --disassemble net/dccp/ipv4.o
```

**Note:** 您需要處於內核樹的頂層以便此獲得您的 C 文件。

如果您無法訪問原始碼，仍然可以使用以下方法調試一些崩潰轉儲（如 Dave Miller 的示例崩潰轉儲輸出所示）：

```
EIP is at +0x14/0x4c0
...
Code: 44 24 04 e8 6f 05 00 00 e9 e8 fe ff ff 8d 76 00 8d bc 27 00 00
00 00 55 57 56 53 81 ec bc 00 00 00 8b ac 24 d0 00 00 00 8b 5d 08
<8b> 83 3c 01 00 00 89 44 24 14 8b 45 28 85 c0 89 44 24 18 0f 85
```

Put the bytes into a "foo.s" file like this:

```
.text
.globl foo
foo:
.byte .... /* bytes from Code: part of OOPS dump */
```

Compile it with "gcc -c -o foo.o foo.s" then look at the output of "objdump --disassemble foo.o".

Output:

```
ip_queue_xmit:
    push    %ebp
    push    %edi
    push    %esi
    push    %ebx
    sub     $0xbc, %esp
    mov     0xd0(%esp), %ebp      ! %ebp = arg0 (skb)
    mov     0x8(%ebp), %ebx      ! %ebx = skb->sk
    mov     0x13c(%ebx), %eax   ! %eax = inet_sk(sk)->opt
```

*scripts/decode decode* 文件可以用來自動完成大部分工作，這取決於正在調試的 CPU 體系結構。

### 報告缺陷

一旦你通過定位缺陷找到了其發生的地方，你可以嘗試自己修復它或者向上游報告它。

為了向上游報告，您應該找出用於開發受影響代碼的郵件列表。這可以使用 `get_maintainer.pl`。

例如，您在 `gspca` 的 `sonixj.c` 文件中發現一個缺陷，則可以通過以下方法找到它的維護者：

```
$ ./scripts/get_maintainer.pl -f drivers/media/usb/gspca/sonixj.c
Hans Verkuil <hverkuil@xs4all.nl> (odd fixer:GSPCA USB WEBCAM DRIVER,commit_signer:1/1=100%)
Mauro Carvalho Chehab <mcchehab@kernel.org> (maintainer:MEDIA INPUT INFRASTRUCTURE (V4L/DVB),
→commit_signer:1/1=100%)
Tejun Heo <tj@kernel.org> (commit_signer:1/1=100%)
Bhaktipriya Shridhar <bhaktipriya96@gmail.com> (commit_signer:1/1=100%,authored:1/1=100%,added_
→lines:4/4=100%,removed_lines:9/9=100%)
linux-media@vger.kernel.org (open list:GSPCA USB WEBCAM DRIVER)
linux-kernel@vger.kernel.org (open list)
```

請注意它將指出：

- 最後接觸原始碼的開發人員（如果這是在 git 樹中完成的）。在上面的例子中是 Tejun 和 Bhaktipriya（在這個特定的案例中，沒有人真正參與這個文件的開發）；
- 驅動維護人員（Hans Verkuil）；
- 子系統維護人員（Mauro Carvalho Chehab）；
- 驅動程序和/或子系統郵件列表（`linux-media@vger.kernel.org`）；
- Linux 內核郵件列表（`linux-kernel@vger.kernel.org`）。

通常，修復缺陷的最快方法是將它報告給用於開發相關代碼的郵件列表（`linux-media ML`），抄送驅動程序維護者（Hans）。

如果你完全不知道該把報告寄給誰，且 `get_maintainer.pl` 也沒有提供任何有用的信息，請發送到 `linux-kernel@vger.kernel.org`。

感謝您的幫助，這使 Linux 儘可能穩定:-)

### 修復缺陷

如果你懂得編程，你不僅可以通過報告錯誤來幫助我們，還可以提供一個解決方案。畢竟，開源就是分享你的工作，你不想因為你的天才而被認可嗎？

如果你決定這樣做，請在制定解決方案後將其提交到上游。

請務必閱讀 `Documentation/process/submitting-patches.rst`，以幫助您的代碼被接受。

## 用 klogd 進行 Oops 跟蹤的注意事項

為了幫助 Linus 和其他內核開發人員，`klogd` 對保護故障的處理提供了大量支持。為了完整支持地址解析，至少應該使用 `sysklogd` 包的 1.3-pl3 版本。

當發生保護故障時，`klogd` 守護進程會自動將內核日誌消息中的重要地址轉換為它們的等效符號。然後通過 `klogd` 使用的任何報告機制來轉發這個已翻譯的內核消息。保護錯誤消息可以直接從消息文件中剪切出來並轉發給內核開發人員。

`klogd` 執行兩種類型的地址解析，靜態翻譯和動態翻譯。靜態翻譯使用 `System.map` 文件。為了進行靜態轉換，`klogd` 守護進程必須能夠在守護進程初始化時找到系統映射文件。有關 `klogd` 如何搜索映射文件的信息，請參見 `klogd` 手冊頁。

當使用內核可加載模塊時，動態地址轉換非常重要。由於內核模塊的內存是從內核的動態內存池中分配的，因此無論是模塊的開頭還是模塊中的函數和符號都沒有固定的位置。

內核支持系統調用，允許程序確定加載哪些模塊及其在內存中的位置。`klogd` 守護進程使用這些系統調用構建了一個符號表，可用於調試可加載內核模塊中發生的保護錯誤。

`klogd` 至少會提供產生保護故障的模塊的名稱。如果可加載模塊的開發人員選擇從模塊導出符號信息，則可能會有其他可用的符號信息。

由於內核模塊環境可以是動態的，因此當模塊環境發生變化時，必須有一種通知 `klogd` 守護進程的機制。有一些可用的命令行選項允許 `klogd` 向當前正在執行的守護進程發出信號示意應該刷新符號信息。有關更多信息，請參閱 `klogd` 手冊頁。

`sysklogd` 發行版附帶了一個補丁，它修改了 `modules-2.0.0` 包，以便在加載或卸載模塊時自動向 `klogd` 發送信號。應用此補丁基本上可無縫支持調試內核可加載模塊發生的保護故障。

以下是 `klogd` 處理的可加載模塊中的保護故障示例：

```
Aug 29 09:51:01 blizard kernel: Unable to handle kernel paging request at virtual address ↴
Aug 29 09:51:01 blizard kernel: current->tss.cr3 = 0062d000, %cr3 = 0062d000
Aug 29 09:51:01 blizard kernel: *pde = 00000000
Aug 29 09:51:01 blizard kernel: Oops: 0002
Aug 29 09:51:01 blizard kernel: CPU:    0
Aug 29 09:51:01 blizard kernel: EIP:    0010:[oops:_oops+16/3868]
Aug 29 09:51:01 blizard kernel: EFLAGS: 00010212
Aug 29 09:51:01 blizard kernel: eax: 315e97cc   ebx: 003a6f80   ecx: 001be77b   edx: 00237c0c
Aug 29 09:51:01 blizard kernel: esi: 00000000   edi: bfffffdb3   ebp: 00589f90   esp: 00589f8c
Aug 29 09:51:01 blizard kernel: ds: 0018   es: 0018   fs: 002b   gs: 002b   ss: 0018
Aug 29 09:51:01 blizard kernel: Process oops_test (pid: 3374, process nr: 21, ↴
Aug 29 09:51:01 blizard kernel: stackpage=00589000)
Aug 29 09:51:01 blizard kernel: Stack: 315e97cc 00589f98 0100b0b4 bfffffed4 0012e38e 00240c64 ↴
Aug 29 09:51:01 blizard kernel: ↴003a6f80 00000001
Aug 29 09:51:01 blizard kernel:      00000000 00237810 bfffff00 0010a7fa 00000003 00000001 ↴
Aug 29 09:51:01 blizard kernel: ↴00000000 bfffff00
Aug 29 09:51:01 blizard kernel:      bfffffdb3 bfffffed4 ffffffd4 0000002b 0007002b 0000002b ↴
Aug 29 09:51:01 blizard kernel: ↴0000002b 00000036
```

```
Aug 29 09:51:01 blizard kernel: Call Trace: [oops:_oops_ioctl+48/80] [_sys_ioctl+254/272] [_  
→system_call+82/128]  
Aug 29 09:51:01 blizard kernel: Code: c7 00 05 00 00 00 eb 08 90 90 90 90 90 90 90 90 90 89 ec 5d  
→c3
```

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:src.res@email.cn)。

**Original** ../../admin-guide/bug-bisect

譯者 吳 想 成 Wu XiangCheng <bobwxc@email.cn> 胡 皓 文 Hu Haowen  
<src.res@email.cn>

### 二分 (bisect) 缺陷

(英文版) 最後更新：2016 年 10 月 28 日

引言

始終嘗試由來自 [kernel.org](http://kernel.org) 的原始碼構建的最新內核。如果您沒有信心這樣做，請將錯誤報告給您的發行版供應商，而不是內核開發人員。

找到缺陷（bug）並不總是那麼容易，不過仍然得去找。如果你找不到它，不要放棄。儘可能多的向相關維護人員報告您發現的信息。請參閱 `Maintainers` 文件以了解您所關注的子系統的維護人員。

在提交錯誤報告之前，請閱讀 [Documentation/admin-guide/reporting-issues.rst](#)。

## 設備未出現 (Devices not appearing)

這通常是由 udev/systemd 引起的。在將其歸咎於內核之前先檢查一下。

### 查找導致缺陷的補丁

使用 git 提供的工具可以很容易地找到缺陷，只要缺陷是可復現的。

操作步驟：

- 從 git 原始碼構建內核
- 以此開始二分<sup>1</sup>:

```
$ git bisect start
```

- 標記損壞的變更集:

```
$ git bisect bad [commit]
```

- 標記正常工作的變更集:

```
$ git bisect good [commit]
```

- 重新構建內核並測試
- 使用以下任一與 git bisect 進行交互:

```
$ git bisect good
```

或:

```
$ git bisect bad
```

這取決於您測試的變更集上是否有缺陷

- 在一些交互之後，git bisect 將給出可能導致缺陷的變更集。
- 例如，如果您知道當前版本有問題，而 4.8 版本是正常的，則可以執行以下操作:

```
$ git bisect start
$ git bisect bad          # Current version is bad
$ git bisect good v4.8
```

如需進一步參考，請閱讀：

- [git-bisect 的手冊頁](#)
- [Fighting regressions with git bisect \(用 git bisect 解決回歸\)](#)

<sup>1</sup> 您可以（可選地）在開始 git bisect 的時候提供 good 或 bad 參數 git bisect start [BAD] [GOOD]

- Fully automated bisecting with “git bisect run” (使用 git bisect run 來全自動二分)
- Using Git bisect to figure out when brokenness was introduced (使用 Git 二分來找出何時引入了錯誤)

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

### Original .../admin-guide/tainted-kernels

譯者 吳 想 成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)> 胡 皓 文 Hu Haowen  
<[src.res@email.cn](mailto:<src.res@email.cn>)>

## 受汙染的內核

當發生一些在稍後調查問題時可能相關的事件時，內核會將自己標記為「受汙染 (tainted)」的。不用太過擔心，大多數情況下運行受汙染的內核沒有問題；這些信息主要在有人想調查某個問題時才有意義的，因為問題的真正原因可能是導致內核受汙染的事件。這就是為什麼來自受汙染內核的缺陷報告常常被開發人員忽略，因此請嘗試用未受汙染的內核重現問題。

請注意，即使在您消除導致汙染的原因（亦即卸載專有內核模塊）之後，內核仍將保持汙染狀態，以表示內核仍然不可信。這也是為什麼內核在注意到內部問題（「kernel bug」）、可恢復錯誤（「kernel oops」）或不可恢復錯誤（「kernel panic」）時會列印受汙染狀態，並將有關此的調試信息寫入日誌 `dmesg` 輸出。也可以通過 `/proc/` 中的文件在運行時檢查受汙染的狀態。

## BUG、Oops 或 Panics 消息中的汙染標誌

在頂部以「CPU:」開頭的一行中可以找到受汙染的狀態；內核是否受到汙染和原因會顯示在進程 ID（「PID:」）和觸發事件命令的縮寫名稱（「Comm:」）之後：

```
BUG: unable to handle kernel NULL pointer dereference at 0000000000000000
Oops: 0002 [#1] SMP PTI
CPU: 0 PID: 4424 Comm: insmod Tainted: P          W 0        4.20.0-0.rc6.fc30 #1
Hardware name: Red Hat KVM, BIOS 0.5.1 01/01/2011
RIP: 0010:my_oops_init+0x13/0x1000 [kpanic]
[...]
```

如果內核在事件發生時沒有被汙染，您將在那裡看到「Not-tainted:」；如果被汙染，那麼它將是「Tainted:」以及字母或空格。在上面的例子中，它看起來是這樣的：

Tainted: P	W	0
------------	---	---

下表解釋了這些字符的含義。在本例中，由於加載了專有模塊 (P)，出現了警告 (W)，並且加載了外部構建的模塊 (0)，所以內核早些時候受到了汙染。要解碼其他字符，請使用下表。

### 解碼運行時的汙染狀態

在運行時，您可以通過讀取 `cat /proc/sys/kernel/tainted` 來查詢受汙染狀態。如果返回 0，則內核沒有受到汙染；任何其他數字都表示受到汙染的原因。解碼這個數字的最簡單方法是使用腳本 `tools/debugging/kernel-chktaint`，您的發行版可能會將其作為名為 `linux-tools` 或 `kernel-tools` 的包的一部分提供；如果沒有，您可以從 [git.kernel.org](https://git.kernel.org) 網站下載此腳本並用 `sh kernel-chktaint` 執行，它會在上面引用的日誌中有類似語句的機器上列印這樣的內容：

```
Kernel is Tainted for following reasons:
 * Proprietary module was loaded (#0)
 * Kernel issued warning (#9)
 * Externally-built ('out-of-tree') module was loaded (#12)
See Documentation/admin-guide/tainted-kernels.rst in the Linux kernel or
https://www.kernel.org/doc/html/latest/admin-guide/tainted-kernels.html for
a more details explanation of the various taint flags.
Raw taint value as int/string: 4609/'P      W 0      '
```

你也可以試著自己解碼這個數字。如果內核被汙染的原因只有一個，那麼這很簡單，在本例中您可以通過下表找到數字。如果你需要解碼有多個原因的數字，因為它是一個位域 (bitfield)，其中每個位表示一個特定類型的汙染的存在或不存在，最好讓前面提到的腳本來處理。但是如果需要快速看一下，可以使用這個 shell 命令來檢查設置了哪些位：

```
$ for i in $(seq 18); do echo $((($i-1)) $(((cat /proc/sys/kernel/tainted)>>($i-1)&1));done
```

## 汙染狀態代碼表

位	日誌	數字	內核被汙染的原因
0	G/P	1	已加載專用模塊
1	_F	2	模塊被強制加載
2	_S	4	內核運行在不合規範的系統上
3	_R	8	模塊被強制卸載
4	_M	16	處理器報告了機器檢測異常 (MCE)
5	_B	32	引用了錯誤的頁或某些意外的頁標誌
6	_U	64	用戶空間應用程式請求的汙染
7	_D	128	內核最近死機了，即曾出現 OOPS 或 BUG
8	_A	256	ACPI 表被用戶覆蓋
9	_W	512	內核發出警告
10	_C	1024	已加載 staging 驅動程序
11	_I	2048	已應用平台固件缺陷的解決方案
12	_O	4096	已加載外部構建（「樹外」）模塊
13	_E	8192	已加載未簽名的模塊
14	_L	16384	發生軟鎖定
15	_K	32768	內核已實時打補丁
16	_X	65536	備用汙染，為發行版定義並使用
17	_T	131072	內核是用結構隨機化插件構建的

註：字符 \_ 表示空白，以便於閱讀表。

## 汙染的更詳細解釋

- 0) G 加載的所有模塊都有 GPL 或兼容許可證，P 加載了任何專有模塊。沒有 MODULE\_LICENSE (模塊許可證) 或 MODULE\_LICENSE 未被 insmod 認可為 GPL 兼容的模塊被認為是專有的。
- 1) F 任何模塊被 insmod -f 強制加載，'' 所有模塊正常加載。
- 2) S 內核運行在不合規範的處理器或系統上：硬體已運行在不受支持的配置中，因此無法保證正確執行。內核將被汙染，例如：
  - 在 x86 上：PAE 是通過 intel CPU (如 Pentium M) 上的 forcepae 強制執行的，這些 CPU 不報告 PAE，但可能有功能實現，SMP 內核在非官方支持的 SMP Athlon CPU 上運行，MSR 被暴露到用戶空間中。
  - 在 arm 上：在某些 CPU (如 Keystone 2) 上運行的內核，沒有啓用某些內核特性。
  - 在 arm64 上：CPU 之間存在不匹配的硬體特性，引導加載程序以不同的模式引導 CPU。
  - 某些驅動程序正在被用在不受支持的體系結構上 (例如 x86\_64 以外的其他系統上的 scsi/snics，非 x86/x86\_64/itanium 上的 scsi/ips，已經損壞了 arm64 上 irqchip/irq-gic 的固件設置…)

- 3) R 模塊被 `rmmod -f` 強制卸載，' ' 所有模塊都正常卸載。
- 4) M 任何處理器報告了機器檢測異常，' ' 未發生機器檢測異常。
- 5) B 頁面釋放函數發現錯誤的頁面引用或某些意外的頁面標誌。這表示硬體問題或內核錯誤；日誌中應該有其他信息指示發生此汙染的原因。
- 6) U 用戶或用戶應用程式特意請求設置受汙染標誌，否則應為 ' '。
- 7) D 內核最近死機了，即出現了 OOPS 或 BUG。
- 8) A ACPI 表被重寫。
- 9) W 內核之前已發出過警告（儘管有些警告可能會設置更具體的汙染標誌）。
- 10) C 已加載 staging 驅動程序。
- 11) I 內核正在處理平台固件（BIOS 或類似軟體）中的嚴重錯誤。
- 12) O 已加載外部構建（「樹外」）模塊。
- 13) E 在支持模塊簽名的內核中加載了未簽名的模塊。
- 14) L 系統上先前發生過軟鎖定。
- 15) K 內核已經實時打了補丁。
- 16) X 備用汙染，由 Linux 發行版定義和使用。
- 17) T 內核構建時使用了 randstruct 插件，它可以有意生成非常不尋常的內核結構布局（甚至是性能病態的布局），這在調試時非常有用。於構建時設置。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

**Original** [.../.../admin-guide/init](https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/admin-guide/init.rst)

譯者 吳 想 成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)> 胡 皓 文 Hu Haowen  
<[src.res@email.cn](mailto:<src.res@email.cn>)>

### 解釋「No working init found.」啓動掛起消息

作者 Andreas Mohr <andi at lisas period de>

Cristian Souza <cristianmsbr at gmail period com>

本文檔提供了加載初始化二進位 (init binary) 失敗的一些高層級原因 (大致按執行順序列出)。

- 1) 無法掛載根文件系統 **Unable to mount root FS** : 請設置「debug」內核參數 (在引導加載程序 bootloader 配置文件或 CONFIG\_CMDLINE) 以獲取更詳細的內核消息。
- 2) 初始二進位不存在於根文件系統上 **init binary doesn't exist on rootfs** : 確保您的根文件系統類型正確 (並且 root= 內核參數指向正確的分區)；擁有所需的驅動程序，例如 SCSI 或 USB 等存儲硬體；文件系統 (ext3、jffs2 等) 是內建的 (或者作為模塊由 initrd 預加載)。
- 3) 控制台設備損壞 **Broken console device** : console= setup 中可能存在衝突 -> 初始控制台不可用 (initial console unavailable)。例如，由於串行 IRQ 問題 (如缺少基於中斷的配置) 導致的某些串行控制台不可靠。嘗試使用不同的 console= device 或像 netconsole=。
- 4) 二進位存在但依賴項不可用 **Binary exists but dependencies not available** : 例如初始化二進位的必需庫依賴項，像 /lib/ld-linux.so.2 丟失或損壞。使用 readelf -d <INIT>|grep NEEDED 找出需要哪些庫。
- 5) 無法加載二進位 **Binary cannot be loaded** : 請確保二進位的體系結構與您的硬體匹配。例如 i386 不匹配 x86\_64，或者嘗試在 ARM 硬體上加載 x86。如果您嘗試在此處加載非二進位文件 (shell 腳本?)，您應該確保腳本在其工作頭 (shebang header) 行 #!/... 中指定能正常工作的解釋器 (包括其庫依賴項)。在處理腳本之前，最好先測試一個簡單的非腳本二進位文件，比如 /bin/sh，並確認它能成功執行。要了解更多信息，請將代碼添加到 init/main.c 以顯示 kernel\_execve() 的返回值。

當您發現新的失敗原因時，請擴展本解釋 (畢竟加載初始化二進位是一個關鍵且艱難的過渡步驟，需要儘可能無痛地進行)，然後向 LKML 提交一個補丁。

待辦事項：

- 通過一個可以存儲 kernel\_execve() 結果值的結構體數組實現各種 run\_init\_process() 調用，並在失敗時通過疊代 所有結果來記錄一切 (非常重要的可用性修復)。
- 試著使實現本身在一般情況下更有幫助，例如在受影響的地方提供額外的錯誤消息。

Todolist:

reporting-bugs ramoops dynamic-debug-howto kdump/index perf/index

這是應用程式開發人員感興趣的章節的開始。可以在這裡找到涵蓋內核 ABI 各個方面的文檔。

Todolist:

sysfs-rules

本手冊的其餘部分包括各種指南，介紹如何根據您的喜好配置內核的特定行為。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

**Translator** 胡皓文 Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

## 清除 `WARN_ONCE`

`WARN_ONCE` / `WARN_ON_ONCE` / `printk_once` 僅僅列印一次消息。

```
echo 1 > /sys/kernel/debug/clear_warn_once
```

可以清除這種狀態並且再次允許列印一次告警信息，這對於運行測試集後重現問題很有用。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

**Translator** 胡皓文 Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

## CPU 負載

Linux 通過``/proc/stat``和``/proc/uptime``導出各種信息，用戶空間工具如 `top(1)` 使用這些信息計算系統花費在某個特定狀態的平均時間。例如：

```
$ iostat Linux 2.6.18.3-exp (linmac) 02/20/2007
```

```
avg-cpu: %user %nice %system %iowait %steal %idle 10.01 0.00 2.92 5.44 0.00
81.63
```

...

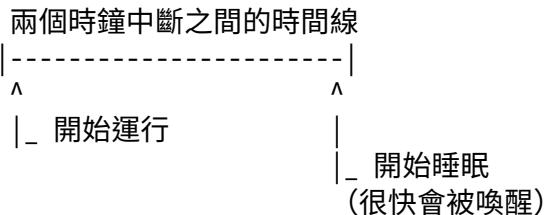
這裡系統認為在默認採樣周期內有 10.01% 的時間工作在用戶空間，2.92% 的時間用在系統空間，總體上有 81.63% 的時間是空閒的。

大多數情況下``/proc/stat``的信息幾乎真實反映了系統信息，然而，由於內核採集這些數據的方式/時間的特點，有時這些信息根本不可靠。

那麼這些信息是如何被搜集的呢？每當時間中斷觸發時，內核查看此刻運行的進程類型，並增加與此類型/狀態進程對應的計數器的值。這種方法的問題是在兩次時間中斷之間系統（進程）能夠在多種狀態之間切換多次，而計數器只增加最後一種狀態下的計數。

舉例—

假設系統有一個進程以如下方式周期性地占用 cpu:



在上面的情況下，根據``/proc/stat``的信息（由於當系統處於空閒狀態時，時間中斷經常會發生）系統的負載將會是 0

大家能夠想像內核的這種行為會發生在許多情況下，這將導致``/proc/stat`` 中存在相當古怪的信息:

```

/* gcc -o hog smallhog.c */
#include <time.h>
#include <limits.h>
#include <signal.h>
#include <sys/time.h>
#define HIST 10

static volatile sig_atomic_t stop;

static void sighandler (int signr)
{
(void) signr;
stop = 1;
}

static unsigned long hog (unsigned long niters)
{
stop = 0;
while (!stop && --niters);
return niters;
}

int main (void)
{
int i;
struct itimerval it = { .it_interval = { .tv_sec = 0, .tv_usec = 1 }, .
.it_value = { .tv_sec = 0, .tv_usec = 1 } };
sigset(SIGALRM, &sighandler);
alarm(1);
for (i = 0; i < HIST; i++)
v[i] = 0;
}

```

```

unsigned long n;
signal (SIGALRM, &sighandler);
setitimer (ITIMER_REAL, &it, NULL);

hog (ULONG_MAX);
for (i = 0; i < HIST; ++i) v[i] = ULONG_MAX - hog (ULONG_MAX);
for (i = 0; i < HIST; ++i) tmp += v[i];
tmp /= HIST;
n = tmp - (tmp / 3.0);

sigemptyset (&set);
sigaddset (&set, SIGALRM);

for (;;) {
    hog (n);
    sigwait (&set, &i);
}
return 0;
}

```

參考—

- <https://lore.kernel.org/r/loom.20070212T063225-663@post.gmane.org>
- Documentation/filesystems/proc.rst (1.8)

謝謝—

Con Kolivas, Pavel Machek

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

**Original** Documentation/admin-guide/unicode.rst

譯者 吳 想 成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)> 胡 皓 文 Hu Haowen  
<[src.res@email.cn](mailto:<src.res@email.cn>)>

### Unicode (統一碼) 支持

(英文版) 上次更新：2005-01-17，版本號 1.4

此文檔由 H. Peter Anvin <[unicode@lanana.org](mailto:unicode@lanana.org)> 管理，是 Linux 註冊名稱與編號管理局（Linux Assigned Names And Numbers Authority，LANANA）項目的一部分。現行版本請見：

<http://www.lanana.org/docs/unicode/admin-guide/unicode.rst>

### 簡介

Linux 內核代碼已被重寫以使用 Unicode 來將字符映射到字體。下載一個 Unicode 到字體（Unicode-to-font）表，八位字符集與 UTF-8 模式都將改用此字體來顯示。

這微妙地改變了八位字符表的語義。現在的四個字符表是：

映射代號	映射名稱	Escape 代碼 (G0)
LAT1_MAP	Latin-1 (ISO 8859-1)	ESC ( B
GRAF_MAP	DEC VT100 pseudographics	ESC ( 0
IBMPC_MAP	IBM code page 437	ESC ( U
USER_MAP	User defined	ESC ( K

特別是 ESC ( U 不再是「直通字體」，因為字體可能與 IBM 字符集完全不同。例如，即使加載了一個 Latin-1 字體，也允許使用塊圖形（block graphics）。

請注意，儘管這些代碼與 ISO 2022 類似，但這些代碼及其用途都與 ISO 2022 不匹配；Linux 有兩個八位代碼（G0 和 G1），而 ISO 2022 有四個七位代碼（G0-G3）。

根據 Unicode 標準/ISO 10646，U+F000 到 U+F8FF 被保留用於作業系統範圍內的分配（Unicode 標準將其稱為「團體區域（Corporate Zone）」，因為這對於 Linux 是不準確的，所以我們稱之為「Linux 區域」）。選擇 U+F000 作為起點，因為它允許直接映射區域以 2 的大倍數開始（以防需要 1024 或 2048 個字符的字體）。這就留下 U+E000 到 U+EFFF 作為最終用戶區。

[v1.2]：Unicodes 範圍從 U+F000 到 U+F7FF 已經被硬編碼為直接映射到加載的字體，繞過了翻譯表。用戶定義的映射現在默認為 U+F000 到 U+F0FF，模擬前述行為。實際上，此範圍可能較短；例如，vgacon 只能處理 256 字符（U+F000..U+F0FF）或 512 字符（U+F000..U+F1FF）字體。

### Linux 區域中定義的實際字符

此外，還定義了 Unicode 1.1.4 中不存在的以下字符；這些字符由 DEC VT 圖形映射使用。[v1.2] 此用法已過時，不應再使用；請參見下文。

U+F800	DEC VT GRAPHICS HORIZONTAL LINE SCAN 1
U+F801	DEC VT GRAPHICS HORIZONTAL LINE SCAN 3
U+F803	DEC VT GRAPHICS HORIZONTAL LINE SCAN 7
U+F804	DEC VT GRAPHICS HORIZONTAL LINE SCAN 9

DEC VT220 使用 6x10 字符矩陣，這些字符在 DEC VT 圖形字符集中形成一個平滑的過渡。我省略了掃描 5 行，因為它也被用作塊圖形字符，因此被編碼為 U+2500 FORMS LIGHT HORIZONTAL。

[v1.3]：這些字符已正式添加到 Unicode 3.2.0 中；它們在 U+23BA、U+23BB、U+23BC、U+23BD 處添加。Linux 現在使用新值。

[v1.2]：添加了以下字符來表示常見的鍵盤符號，這些符號不太可能被添加到 Unicode 中，因為它們非常討厭地取決於特定供應商。當然，這是糟糕設計的一個好例子。

U+F810	KEYBOARD SYMBOL FLYING FLAG
U+F811	KEYBOARD SYMBOL PULLDOWN MENU
U+F812	KEYBOARD SYMBOL OPEN APPLE
U+F813	KEYBOARD SYMBOL SOLID APPLE

## 克林貢 (Klingon) 語支持

1996 年，Linux 是世界上第一個添加對人工語言克林貢支持的作業系統，克林貢是由 Marc Okrand 為《星際迷航》電視連續劇創造的。這種編碼後來被徵募 Unicode 註冊表 (ConScript Unicode Registry, CSUR) 採用，並建議(但最終被拒絕)納入 Unicode 平面一。不過，它仍然是 Linux 區域中的 Linux/CSUR 私有分配。

這種編碼已經得到克林貢語言研究所 (Klingon Language Institute) 的認可。有關更多信息，請聯繫他們：

<http://www.kli.org/>

由於 Linux CZ 開頭部分的字符大多是 dingbats/symbols/forms 類型，而且這是一種語言，因此根據標準 Unicode 慣例，我將它放置在 16 單元的邊界上。

---

**Note:** 這個範圍現在由徵募 Unicode 註冊表正式管理。規範性引用文件為：

<https://www.evertype.com/standards/csur/klingon.html>

---

克林貢語有一個 26 個字符的字母表，一個 10 位數的位置數字書寫系統，從左到右，從上到下書寫。

克林貢字母的幾種字形已經被提出。但是由於這組符號看起來始終是一致的，只有實際的形狀不同，因此按照標準 Unicode 慣例，這些差異被認為是字體變體。

U+F8D0	KLINGON LETTER A
U+F8D1	KLINGON LETTER B
U+F8D2	KLINGON LETTER CH
U+F8D3	KLINGON LETTER D
U+F8D4	KLINGON LETTER E
U+F8D5	KLINGON LETTER GH
U+F8D6	KLINGON LETTER H
U+F8D7	KLINGON LETTER I
U+F8D8	KLINGON LETTER J
U+F8D9	KLINGON LETTER L
U+F8DA	KLINGON LETTER M
U+F8DB	KLINGON LETTER N
U+F8DC	KLINGON LETTER NG
U+F8DD	KLINGON LETTER O
U+F8DE	KLINGON LETTER P
U+F8DF	KLINGON LETTER Q - Written <q> in standard Okrand Latin transliteration
U+F8E0	KLINGON LETTER QH - Written <Q> in standard Okrand Latin transliteration
U+F8E1	KLINGON LETTER R
U+F8E2	KLINGON LETTER S
U+F8E3	KLINGON LETTER T
U+F8E4	KLINGON LETTER TLH
U+F8E5	KLINGON LETTER U
U+F8E6	KLINGON LETTER V
U+F8E7	KLINGON LETTER W
U+F8E8	KLINGON LETTER Y
U+F8E9	KLINGON LETTER GLOTTAL STOP
U+F8F0	KLINGON DIGIT ZERO
U+F8F1	KLINGON DIGIT ONE
U+F8F2	KLINGON DIGIT TWO
U+F8F3	KLINGON DIGIT THREE
U+F8F4	KLINGON DIGIT FOUR
U+F8F5	KLINGON DIGIT FIVE
U+F8F6	KLINGON DIGIT SIX
U+F8F7	KLINGON DIGIT SEVEN
U+F8F8	KLINGON DIGIT EIGHT
U+F8F9	KLINGON DIGIT NINE
U+F8FD	KLINGON COMMA
U+F8FE	KLINGON FULL STOP
U+F8FF	KLINGON SYMBOL FOR EMPIRE

## 其他虛構和人工字母

自從分配了克林貢 Linux Unicode 塊之後，John Cowan <jcowan@reutershealth.com> 和 Michael Everson <everson@everttype.com> 建立了一個虛構和人工字母的註冊表。徵募 Unicode 註冊表請訪問：

<https://www.everttype.com/standards/csur/>

所使用的範圍位於最終用戶區域的低端，因此無法進行規範化分配，但建議希望對虛構字母進行編碼的人員使用這些代碼，以實現互操作性。對於克林貢語，CSUR 採用了 Linux 編碼。CSUR 的人正在推動將 Tengwar 和 Cirth 添加到 Unicode 平面一；將克林貢添加到 Unicode 平面一被拒絕，因此上述編碼仍然是官方的。

Todolist:

```
acpi/index aoe/index auxdisplay/index bcache binderfs binfmt-misc blockdev/index
bootconfig braille-console btmrvl cgroup-v1/index cgroup-v2 cifs/index cputopology
dell_rbu device-mapper/index edid efi-stub ext4 nfs/index gpio/index highuid
hw_random initrd iostats java jfs kernel-per-CPU-kthreads laptops/index lcd-panel-
cgram ldm lockup-watchdogs LSM/index md media/index mm/index module-signing
mono namespaces/index numastat parport perf-security pm/index pnp rapidio ras rtc
serial-console svga sysrq thunderbolt ufs vga-softcursor video-output xfs
```

TODOList:

- kbuild/index

## \* 固件相關文檔

下列文檔描述了內核需要的平台固件相關信息。

TODOList:

- firmware-guide/index
- devicetree/index

## \* 應用程式開發人員文檔

用戶空間 API 手冊涵蓋了描述應用程式開發人員可見內核接口方面的文檔。

TODOList:

- userspace-api/index

### \* 內核開發簡介

這些手冊包含有關如何開發內核的整體信息。內核社區非常龐大，一年下來有數千名開發人員做出貢獻。與任何大型社區一樣，知道如何完成任務將使得更改合併的過程變得更加容易。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

---

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:src.res@email.cn)。

---

**Original** Documentation/process/index.rst

**Translator** Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)> Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

### \* 與 Linux 內核社區一起工作

你想成為 Linux 內核開發人員嗎？歡迎之至！在學習許多關於內核的技術知識的同時，了解我們社區的工作方式也很重要。閱讀這些文檔可以讓您以更輕鬆的、麻煩更少的方式將更改合併到內核。

以下是每位開發人員都應閱讀的基本指南：

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

---

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:src.res@email.cn)。

---

**Original** Documentation/process/howto.rst

譯者：

英文版維護者： Greg Kroah-Hartman <[greg@kroah.com](mailto:greg@kroah.com)>  
中文版維護者： 李陽 Li Yang <[leoyang.li@nxp.com](mailto:leoyang.li@nxp.com)>  
中文版翻譯者： 李陽 Li Yang <[leoyang.li@nxp.com](mailto:leoyang.li@nxp.com)>  
時奎亮 Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>  
中文版校譯者：  
    鍾宇 TripleX Chung <[xxx.phy@gmail.com](mailto:xxx.phy@gmail.com)>

陳琦 Maggie Chen <chenqi@beyondsoft.com>  
 王聰 Wang Cong <xifyou.wangcong@gmail.com>  
 胡皓文 Hu Haowen <src.res@email.cn>

## 如何參與 Linux 內核開發

這是一篇將如何參與 Linux 內核開發的相關問題一網打盡的終極祕笈。它將指導你成為一名 Linux 內核開發者，並且學會如何同 Linux 內核開發社區合作。它儘可能不包括任何關於內核編程的技術細節，但會給你指引一條獲得這些知識的正確途徑。

如果這篇文章中的任何內容不再適用，請給文末列出的文件維護者發送補丁。

## 入門

你想了解如何成為一名 Linux 內核開發者？或者老闆吩咐你「給這個設備寫個 Linux 驅動程序」？這篇文章的目的就是教會你達成這些目標的全部訣竅，它將描述你需要經過的流程以及給出如何同內核社區合作的一些提示。它還將試圖解釋內核社區為何這樣運作。

Linux 內核大部分是由 C 語言寫成的，一些體系結構相關的代碼用到了彙編語言。要參與內核開發，你必須精通 C 語言。除非你想為某個架構開發底層代碼，否則你並不需要了解（任何體系結構的）彙編語言。下面列舉的書籍雖然不能替代紮實的 C 語言教育和多年的開發經驗，但如果需要的話，做為參考還是不錯的：

- “The C Programming Language” by Kernighan and Ritchie [Prentice Hall] 《C 程序設計語言（第 2 版·新版）》（徐寶文李志譯）[機械工業出版社]
- “Practical C Programming” by Steve Oualline [O’ Reilly] 《實用 C 語言編程（第三版）》（郭大海譯）[中國電力出版社]
- “C: A Reference Manual” by Harbison and Steele [Prentice Hall] 《C 語言參考手冊（原書第 5 版）》（邱仲潘等譯）[機械工業出版社]

Linux 內核使用 GNU C 和 GNU 工具鏈開發。雖然它遵循 ISO C89 標準，但也用到了一些標準中沒有定義的擴展。內核是自給自足的 C 環境，不依賴於標準 C 庫的支持，所以並不支持 C 標準中的部分定義。比如 long long 類型的大數除法和浮點運算就不允許使用。有時候確實很難弄清楚內核對工具鏈的要求和它所使用的擴展，不幸的是目前還沒有明確的參考資料可以解釋它們。請查閱 gcc 信息頁（使用「info gcc」命令顯示）獲得一些這方面信息。

請記住你是在學習怎麼和已經存在的開發社區打交道。它由一羣形形色色的人組成，他們對代碼、風格和過程有著很高的標準。這些標準是在長期實踐中總結出來的，適應於地理上分散的大型開發團隊。它們已經被很好得整理成檔，建議你在開發之前儘可能多的學習這些標準，而不要期望別人來適應你或者你公司的行為方式。

### 法律問題

Linux 內核原始碼都是在 GPL（通用公共許可證）的保護下發布的。要了解這種許可的細節請查看原始碼主目錄下的 COPYING 文件。Linux 內核許可準則和如何使用 SPDX <<https://spdx.org/>> 標誌符說明在這個文件中 *Documentation/translations/zh\_TW/process/license-rules.rst* 如果你對它還有更深入問題請聯繫律師，而不要在 Linux 內核郵件組上提問。因為郵件組裡的人並不是律師，不要期望他們的話有法律效力。

對於 GPL 的常見問題和解答，請訪問以下連結：<https://www.gnu.org/licenses/gpl-faq.html>

### 文檔

Linux 內核代碼中包含有大量的文檔。這些文檔對於學習如何與內核社區互動有著不可估量的價值。當一個新的功能被加入內核，最好把解釋如何使用這個功能的文檔也放進內核。當內核的改動導致面向用戶空間的接口發生變化時，最好將相關信息或手冊頁 (manpages) 的補丁發到 [mtk.manpages@gmail.com](mailto:mtk.manpages@gmail.com)，以向手冊頁 (manpages) 的維護者解釋這些變化。

以下是內核代碼中需要閱讀的文檔：

**Documentation/admin-guide/README.rst** 文件簡要介紹了 Linux 內核的背景，並且描述了如何配置和編譯內核。內核的新用戶應該從這裡開始。

**Documentation/process/changes.rst** 文件給出了用來編譯和使用內核所需要的最小軟體包列表。

**Documentation/translations/zh\_TW/process/coding-style.rst** 描述 Linux 內核的代碼風格和理由。所有新代碼需要遵守這篇文檔中定義的規範。大多數維護者只會接收符合規定的補丁，很多人也只會幫忙檢查符合風格的代碼。

*Documentation/translations/zh\_TW/process/submitting-patches.rst*  
*Documentation/process/submitting-drivers.rst*

這兩份文檔明確描述如何創建和發送補丁，其中包括（但不僅限於）：

- 郵件內容
- 郵件格式
- 選擇收件人

遵守這些規定並不能保證提交成功（因為所有補丁需要通過嚴格的內容和風格審查），但是忽視他們幾乎就意味著失敗。

其他關於如何正確地生成補丁的優秀文檔包括：“The Perfect Patch”

<https://www.ozlabs.org/~akpm/stuff/tpp.txt>

“Linux kernel patch submission format”

<https://web.archive.org/web/20180829112450/http://linux.yyz.us/patch-format.html>

**Documentation/translations/zh\_TW/process/stable-api-nonsense.rst** 論證內核為什麼特意不包括穩定的內核內部 API，也就是說不包括像這樣的特性：

- 子系統中間層（為了兼容性？）
- 在不同作業系統間易於移植的驅動程序
- 減緩（甚至阻止）內核代碼的快速變化

這篇文檔對於理解 Linux 的開發哲學至關重要。對於將開發平台從其他操作系統轉移到 Linux 的人來說也很重要。

**Documentation/admin-guide/security-bugs.rst** 如果你認為自己發現了 Linux 內核的安全性問題，請根據這篇文檔中的步驟來提醒其他內核開發者並幫助解決這個問題。

*Documentation/translations/zh\_TW/process/management-style.rst*

描述內核維護者的工作方法及其共有特點。這對於剛剛接觸內核開發（或者對它感到好奇）的人來說很重要，因為它解釋了很多對於內核維護者獨特行為的普遍誤解與迷惑。

**Documentation/process/stable-kernel-rules.rst** 解釋了穩定版內核發布的規則，以及如何將改動放入這些版本的步驟。

**Documentation/process/kernel-docs.rst** 有助於內核開發的外部文檔列表。如果你在內核自帶的文檔中沒有找到你想找的內容，可以查看這些文檔。

**Documentation/process/applying-patches.rst** 關於補丁是什麼以及如何將它打在不同內核開發分支上的好介紹

內核還擁有大量從代碼自動生成或者從 ReStructuredText(ReST) 標記生成的文檔，比如這個文檔，它包含內核內部 API 的全面介紹以及如何妥善處理加鎖的規則。所有這些文檔都可以通過運行以下命令從內核代碼中生成為 PDF 或 HTML 文檔：

```
make pdfdocs  
make htmldocs
```

ReST 格式的文檔會生成在 Documentation/output. 目錄中。它們也可以用下列命令生成 LaTeX 和 ePub 格式文檔：

```
make latexdocs  
make epubdocs
```

### 如何成為內核開發者

如果你對 Linux 內核開發一無所知，你應該訪問「Linux 內核新手」計劃：

<https://kernelnewbies.org>

它擁有一個可以問各種最基本的內核開發問題的郵件列表（在提問之前一定要記得查找已往的郵件，確認是否有人已經回答過相同的問題）。它還擁有一個可以獲得實時反饋的 IRC 聊天頻道，以及大量對於學習 Linux 內核開發相當有幫助的文檔。

網站簡要介紹了原始碼組織結構、子系統劃分以及目前正在進行的項目（包括內核中的和單獨維護的）。它還提供了一些基本的幫助信息，比如如何編譯內核和打補丁。

如果你想加入內核開發社區並協助完成一些任務，卻找不到從哪裡開始，可以訪問「Linux 內核房管員」計劃：

<https://kernelnewbies.org/KernelJanitors>

這是極佳的起點。它提供一個相對簡單的任務列表，列出內核代碼中需要被重新整理或者改正的地方。通過和負責這個計劃的開發者們一同工作，你會學到將補丁集成進內核的基本原理。如果還沒有決定下一步要做什么的話，你還可能會得到方向性的指點。

在真正動手修改內核代碼之前，理解要修改的代碼如何運作是必需的。要達到這個目的，沒什麼辦法比直接讀代碼更有效了（大多數花招都會有相應的注釋），而且一些特製的工具還可以提供幫助。例如，「Linux 代碼交叉引用」項目就是一個值得特別推薦的幫助工具，它將原始碼顯示在有編目和索引的網頁上。其中一個更新及時的內核源碼庫，可以通過以下地址訪問：

<https://elixir.bootlin.com/>

### 開發流程

目前 Linux 內核開發流程包括幾個「主內核分支」和很多子系統相關的內核分支。這些分支包括：

- Linus 的內核源碼樹
- 多個主要版本的穩定版內核樹
- 子系統相關的內核樹
- linux-next 集成測試樹

## 主線樹

主線樹是由 Linus Torvalds 維護的。你可以在 <https://kernel.org> 網站或者代碼庫中下找到它。它的開發遵循以下步驟：

- 每當一個新版本的內核被發布，為期兩周的集成窗口將被打開。在這段時間裡維護者可以向 Linus 提交大段的修改，通常這些修改已經被放到-mm 內核中幾個星期了。提交大量修改的首選方式是使用 git 工具（內核的代碼版本管理工具，更多的信息可以在 <https://git-scm.com/> 獲取），不過使用普通補丁也是可以的。
- 兩個星期以後-rc1 版本內核發布。之後只有不包含可能影響整個內核穩定性的新功能的補丁才可能被接受。請注意一個全新的驅動程序（或者文件系統）有可能在-rc1 後被接受是因為這樣的修改完全獨立，不會影響其他的代碼，所以沒有造成內核退步的風險。在-rc1 以後也可以用 git 向 Linus 提交補丁，不過所有的補丁需要同時被發送到相應的公眾郵件列表以徵詢意見。
- 當 Linus 認為當前的 git 源碼樹已經達到一個合理健全的狀態足以發布供人測試時，一個新的-rc 版本就會被發布。計劃是每周都發布新的-rc 版本。
- 這個過程一直持續下去直到內核被認為達到足夠穩定的狀態，持續時間大概是 6 個星期。

關於內核發布，值得一提的是 Andrew Morton 在 **linux-kernel** 電郵列表中如是說：「沒有人知道新內核何時會被發布，因為發布是根據已知 bug 的情況來決定的，而不是根據一個事先制定好的時間表。」

## 子系統特定樹

各種內核子系統的維護者——以及許多內核子系統開發人員——在原始碼庫中公開了他們當前的開發狀態。這樣，其他人就可以看到內核的不同區域發生了什麼。在開發速度很快的領域，可能會要求開發人員將提交的內容建立在這樣的子系統內核樹上，這樣就避免了提交與其他已經進行的工作之間的衝突。

這些存儲庫中的大多數都是 Git 樹，但是也有其他的 scm 在使用，或者補丁隊列被發布為 Quilt 系列。這些子系統存儲庫的地址列在 MAINTAINERS 文件中。其中許多可以在 <https://git.kernel.org/> 上瀏覽。

在將一個建議的補丁提交到這樣的子系統樹之前，需要對它進行審查，審查主要發生在郵件列表上（請參見下面相應的部分）。對於幾個內核子系統，這個審查過程是通過工具補丁跟蹤的。Patchwork 提供了一個 Web 界面，顯示補丁發布、對補丁的任何評論或修訂，維護人員可以將補丁標記為正在審查、接受或拒絕。大多數補丁網站都列在 <https://patchwork.kernel.org/>

## Linux-next 集成測試樹

在將子系統樹的更新合併到主線樹之前，需要對它們進行集成測試。為此，存在一個特殊的測試存儲庫，其中幾乎每天都會提取所有子系統樹：

<https://git.kernel.org/?p=linux/kernel/git/next/linux-next.git>

通過這種方式，Linux-next 對下一個合併階段將進入主線內核的內容給出了一個概要展望。非常喜歡冒險的測試者運行測試 Linux-next。

### 多個主要版本的穩定版內核樹

由 3 個數字組成的內核版本號說明此內核是-stable 版本。它們包含內核的相對較小且至關重要的修補，這些修補針對安全性問題或者嚴重的內核退步。

這種版本的內核適用於那些期望獲得最新的穩定版內核並且不想參與測試開發版或者實驗版的用戶。

穩定版內核樹版本由「穩定版」小組（郵件地址 <[stable@vger.kernel.org](mailto:stable@vger.kernel.org)>）維護，一般隔周發布新版本。

內核源碼中的 Documentation/process/stable-kernel-rules.rst 文件具體描述了可被穩定版內核接受的修改類型以及發布的流程。

### 報告 bug

[bugzilla.kernel.org](http://bugzilla.kernel.org) 是 Linux 內核開發者們用來跟蹤內核 Bug 的網站。我們鼓勵用戶在這個工具中報告找到的所有 bug。如何使用內核 bugzilla 的細節請訪問：

<http://test.kernel.org/bugzilla/faq.html>

內核源碼主目錄中的 :ref:`admin-guide/reporting-bugs.rst <reportingbugs>` 文件里有一個很好的模板。它指導用戶如何報告可能的內核 bug 以及需要提供哪些信息來幫助內核開發者們找到問題的根源。

### 利用 bug 報告

練習內核開發技能的最好辦法就是修改其他人報告的 bug。你不光可以幫助內核變得更加穩定，還可以學會如何解決實際問題從而提高自己的技能，並且讓其他開發者感受到你的存在。修改 bug 是贏得其他開發者讚譽的最好辦法，因為並不是很多人都喜歡浪費時間去修改別人報告的 bug。

要嘗試修改已知的 bug，請訪問 <http://bugzilla.kernel.org> 網址。

### 郵件列表

正如上面的文檔所描述，大多數的骨幹內核開發者都加入了 Linux Kernel 郵件列表。如何訂閱和退訂列表的細節可以在這裡找到：

<http://vger.kernel.org/vger-lists.html#linux-kernel>

網上很多地方都有這個郵件列表的存檔 (archive)。可以使用搜尋引擎來找到這些存檔。比如：

<http://dir.gmane.org/gmane.linux.kernel>

在發信之前，我們強烈建議你先在存檔中搜索你想要討論的問題。很多已經被詳細討論過的問題只在郵件列表的存檔中可以找到。

大多數內核子系統也有自己獨立的郵件列表來協調各自的開發工作。從 MAINTAINERS 文件中可以找到不同話題對應的郵件列表。

很多郵件列表架設在 kernel.org 伺服器上。這些列表的信息可以在這裡找到：

<http://vger.kernel.org/vger-lists.html>

在使用這些郵件列表時，請記住保持良好的行為習慣。下面的連結提供了與這些列表（或任何其它郵件列表）交流的一些簡單規則，雖然內容有點濫竽充數。

<http://www.albion.com/netiquette/>

當有很多人回覆你的郵件時，郵件的抄送列表會變得很長。請不要將任何人從抄送列表中刪除，除非你有足夠的理由這麼做。也不要只回復到郵件列表。請習慣於同一封郵件接收兩次（一封來自發送者一封來自郵件列表），而不要試圖通過添加一些奇特的郵件頭來解決這個問題，人們不會喜歡的。

記住保留你所回復內容的上下文和源頭。在你回覆郵件的頂部保留「某某某說到……」這幾行。將你的評論加在被引用的段落之間而不要放在郵件的頂部。

如果 你 在 郵 件 中 附 帶 補 丁， 請 確 認 它 們 是 可 以 直 接 閱 讀 的 純 文 本  
(如*Documentation/translations/zh\_TW/process/submitting-patches.rst* 文檔中所述)。內核開  
發者們不希望遇到附件或者被壓縮了的補丁。只有這樣才能保證他們可以直接評論你的每行代碼。請確保你  
使用的郵件發送程序不會修改空格和制表符。一個防範性的測試方法是先將郵件發送給自己，然後自己嘗試  
是否可以順利地打上收到的補丁。如果測試不成功，請調整或者更換你的郵件發送程序直到它正確工作為止。

總而言之，請尊重其他的郵件列表訂閱者。

## 同內核社區合作

內核社區的目標就是提供盡善盡美的內核。所以當你提交補丁期望被接受進內核的時候，它的技術價值以及其他方面都將被評審。那麼你可能會得到什麼呢？

- 批評
- 評論
- 要求修改
- 要求證明修改的必要性
- 沉默

要記住，這些是把補丁放進內核的正常情況。你必須學會聽取對補丁的批評和評論，從技術層面評估它們，然後要麼重寫你的補丁要麼簡明扼要地論證修改是不必要的。如果你發的郵件沒有得到任何回應，請過幾天後再試一次，因為有時信件會湮沒在茫茫信海中。

你不應該做的事情：

- 期望自己的補丁不受任何質疑就直接被接受
- 翻臉
- 忽略別人的評論
- 沒有按照別人的要求做任何修改就重新提交

在一個努力追尋最好技術方案的社區里，對於一個補丁有多少好處總會有不同的見解。你必須要抱著合作的態度，願意改變自己的觀點來適應內核的風格。或者至少願意去證明你的想法是有價值的。記住，犯錯誤是允許的，只要你願意朝著正確的方案去努力。

如果你的第一個補丁換來的是一堆修改建議，這是很正常的。這並不代表你的補丁不會被接受，也不意味著有人和你作對。你只需要改正所有提出的問題然後重新發送你的補丁。

### 內核社區和公司文化的差異

內核社區的工作模式同大多數傳統公司開發隊伍的工作模式並不相同。下面這些例子，可以幫助你避免某些可能發生問題：用這些話介紹你的修改提案會有好處：

- 它同時解決了多個問題
- 它刪除了 2000 行代碼
- 這是補丁，它已經解釋了我想要說明的
- 我在 5 種不同的體系結構上測試過它……
- 這是一系列小補丁用來……
- 這個修改提高了普通機器的性能……

應該避免如下的說法：

- 我們在 AIX/ptx/Solaris 就是這麼做的，所以這麼做肯定是好的……
- 我做這行已經 20 年了，所以……
- 為了我們公司賺錢考慮必須這麼做
- 這是我們的企業產品線所需要的
- 這裡是描述我觀點的 1000 頁設計文檔
- 這是一個 5000 行的補丁用來……
- 我重寫了現在亂七八糟的代碼，這就是……
- 我被規定了最後期限，所以這個補丁需要立刻被接受

另外一個內核社區與大部分傳統公司的軟體開發隊伍不同的地方是無法面對面地交流。使用電子郵件和 IRC 聊天工具做為主要溝通工具的一個好處是性別和種族歧視將會更少。Linux 內核的工作環境更能接受婦女和少數族羣，因為每個人在別人眼里只是一個郵件地址。國際化也幫助了公平的實現，因為你無法通過姓名來判斷人的性別。男人有可能叫李麗，女人也有可能叫王剛。大多數在 Linux 內核上工作過並表達過看法的女性對在 linux 上工作的經歷都給出了正面的評價。

對於一些不習慣使用英語的人來說，語言可能是一個引起問題的障礙。在郵件列表中要正確地表達想法必需良好地掌握語言，所以建議你在發送郵件之前最好檢查一下英文寫得是否正確。

## 拆分修改

Linux 內核社區並不喜歡一下接收大段的代碼。修改需要被恰當地介紹、討論並且拆分成獨立的小段。這幾乎完全和公司中的習慣背道而馳。你的想法應該在開發最開始的階段就讓大家知道，這樣你就可以及時獲得對你正在進行的開發的反饋。這樣也會讓社區覺得你是在和他們協作，而不是僅僅把他們當作傾銷新功能的對象。無論如何，你不要一次性地向郵件列表發送 50 封信，你的補丁序列應該永遠用不到這麼多。

將補丁拆開的原因如下：

- 1) 小的補丁更有可能被接受，因為它們不需要太多的時間和精力去驗證其正確性。一個 5 行的補丁，可能在維護者看了一眼以後就會被接受。而 500 行的補丁則需要數個小時來審查其正確性（所需時間隨補丁大小增加大約呈指數級增長）。

當出了問題的時候，小的補丁也會讓調試變得非常容易。一個一個補丁地回溯將會比仔細剖析一個被打上的大補丁（這個補丁破壞了其他東西）容易得多。

- 2) 不光發送小的補丁很重要，在提交之前重新編排、化簡（或者僅僅重新排列）補丁也是很重要的。

這裡有內核開發者 **Al Viro** 打的一個比方：「想像一個老師正在給學生批改數學作業。老師並不希望看到學生為了得到正確解法所進行的嘗試和產生的錯誤。他希望看到的是最乾淨最優雅的解答。好學生了解這點，絕不會把最終解決之前的中間方案提交上去。」

內核開發也是這樣。維護者和評審者不希望看到一個人在解決問題時的思考過程。他們只希望看到簡單和優雅的解決方案。

直接給出一流的解決方案，和社區一起協作討論尚未完成的工作，這兩者之間似乎很難找到一個平衡點。所以最好儘早開始收集有利於你進行改進的反饋；同時也要保證修改分成很多小塊，這樣在整個項目都準備好被包含進內核之前，其中的一部分可能會先被接收。

必須了解這樣做是不可接受的：試圖將未完成的工作提交進內核，然後再找時間修復。

## 證明修改的必要性

除了將補丁拆成小塊，很重要的一點是讓 Linux 社區了解他們為什麼需要這樣修改。你必須證明新功能是有人需要的並且是有用的。

## 記錄修改

當你發送補丁的時候，需要特別留意郵件正文的內容。因為這裡的信息將會做為補丁的修改記錄 (ChangeLog)，會被一直保留以備大家查閱。它需要完全地描述補丁，包括：

- 為什麼需要這個修改
- 補丁的總體設計
- 實現細節
- 測試結果

想了解它具體應該看起來像什麼，請查閱以下文檔中的「**ChangeLog**」章節：

「**The Perfect Patch**」 <https://www.ozlabs.org/~akpm/stuff/tpp.txt>

這些事情有時候做起來很難。要在任何方面都做到完美可能需要好幾年時間。這是一個持續提高的過程，它需要大量的耐心和決心。只要不放棄，你一定可以做到。很多人已經做到了，而他們都曾經和現在的你站在同樣的起點上。

### 感謝

感謝 Paolo Ciarrocchi 允許「開發流程」部分基於他所寫的文章 ([http://www.kerneltravel.net/newbie/2.6-development\\_process](http://www.kerneltravel.net/newbie/2.6-development_process))，感謝 Randy Dunlap 和 Gerrit Huizenga 完善了應該說和不該說的列表。感謝 Pat Mochel, Hanna Linder, Randy Dunlap, Kay Sievers, Vojtech Pavlik, Jan Kara, Josh Boyer, Kees Cook, Andrew Morton, Andi Kleen, Vadim Lobanov, Jesper Juhl, Adrian Bunk, Keri Harris, Frans Pop, David A. Wheeler, Junio Hamano, Michael Kerrisk 和 Alex Shepard 的評審、建議和貢獻。沒有他們的幫助，這篇文檔是不可能完成的。

英文版維護者：Greg Kroah-Hartman <[greg@kroah.com](mailto:greg@kroah.com)>

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:src.res@email.cn)。

**Original** Documentation/process/code-of-conduct.rst

**Translator** Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)> Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

### 貢獻者契約行為準則

#### 我們的誓言

為了營造一個開放、友好的環境，我們作為貢獻者和維護人承諾，讓我們的社區和參與者，擁有一個無騷擾的體驗，無論年齡、體型、殘疾、種族、性別特徵、性別認同和表達、經驗水平、教育程度、社會狀況，經濟地位、國籍、個人外貌、種族、宗教或性身份和取向。

## 我們的標準

有助於創造積極環境的行為包括：

- 使用歡迎和包容的語言
- 尊重不同的觀點和經驗
- 優雅地接受建設性的批評
- 關注什麼對社區最有利
- 對其他社區成員表示同情

參與者的不可接受行為包括：

- 使用性意味的語言或意象以及不受歡迎的性注意或者更過分的行為
- 煽動、侮辱/貶損評論以及個人或政治攻擊
- 公開或私下騷擾
- 未經明確許可，發布他人的私人信息，如物理或電子地址。
- 在專業場合被合理認為不適當的其他行為

## 我們的責任

維護人員負責澄清可接受行為的標準，並應針對任何不可接受行為採取適當和公平的糾正措施。

維護人員有權和責任刪除、編輯或拒絕與本行為準則不一致的評論、承諾、代碼、wiki 編輯、問題和其他貢獻，或暫時或永久禁止任何貢獻者從事他們認為不適當、威脅、冒犯或有害的其他行為。

## 範圍

當個人代表項目或其社區時，本行為準則既適用於項目空間，也適用於公共空間。代表一個項目或社區的例子包括使用一個正式的項目電子郵件地址，通過一個正式的社交媒體帳戶發布，或者在線或離線事件中擔任指定的代表。項目維護人員可以進一步定義和澄清項目的表示。

## 執行

如有濫用、騷擾或其他不可接受的行為，可聯繫行為準則委員會 <[conduct@kernel.org](mailto:conduct@kernel.org)>。所有投訴都將接受審查和調查，並將得到必要和適當的答覆。行為準則委員會有義務對事件報告人保密。具體執行政策的進一步細節可單獨公布。

### 歸屬

本行為準則改編自《貢獻者契約》，版本 1.4，可從 <https://www.contributor-covenant.org/version/1/4/code-of-conduct.html> 獲取。

### 解釋

有關 Linux 內核社區如何解釋此文檔，請參閱 [Linux 內核貢獻者契約行為準則解釋](#)

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

**Original** Documentation/process/code-of-conduct-interpretation.rst

**Translator** Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)> Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

### Linux 內核貢獻者契約行為準則解釋

貢獻者契約行為準則 準則是一個通用文檔，旨在為幾乎所有開源社區提供一套規則。每個開源社區都是獨一無二的，Linux 內核也不例外。因此，本文描述了 Linux 內核社區中如何解釋它。我們也不希望這種解釋隨著時間的推移是靜態的，並將根據需要進行調整。

與開發軟體的「傳統」方法相比，Linux 內核開發工作是一個非常個人化的過程。你的貢獻和背後的想法將被仔細審查，往往導致批判和批評。審查將幾乎總是需要改進，材料才能包括在內核中。要知道這是因為所有相關人員都希望看到 Linux 整體成功的最佳解決方案。這個開發過程已經被證明可以創建有史以來最健壯的作業系統內核，我們不想做任何事情來導致提交質量和最終結果的下降。

### 維護者

行為準則多次使用「維護者」一詞。在內核社區中，「維護者」是負責子系統、驅動程序或文件的任何人，並在內核原始碼樹的維護者文件中列出。

## 責任

《行為準則》提到了維護人員的權利和責任，這需要進一步澄清。

首先，最重要的是，有一個合理的期望是由維護人員通過實例來領導。

也就是說，我們的社區是廣闊的，對維護者沒有新的要求，他們單方面處理其他人在他們活躍的社區的行為。這一責任由我們所有人承擔，最終《行為準則》記錄了最終的上訴路徑，以防有關行為問題的問題懸而未決。

維護人員應該願意在出現問題時提供幫助，並在需要時與社區中的其他人合作。如果您不確定如何處理出現的情況，請不要害怕聯繫技術諮詢委員會（TAB）或其他維護人員。除非您願意，否則不會將其視為違規報告。如果您不確定是否該聯繫 TAB 或任何其他維護人員，請聯繫我們的衝突調解人 Mishi Choudhary <[mishi@linux.com](mailto:mishi@linux.com)>。

最後，「善待對方」才是每個人的最終目標。我們知道每個人都是人，有時我們都會失敗，但我們所有人的首要目標應該是努力友好地解決問題。執行行為準則將是最後的選擇。

我們的目標是創建一個強大的、技術先進的作業系統，以及所涉及的技術複雜性，這自然需要專業知識和決策。

所需的專業知識因貢獻領域而異。它主要由上下文和技術複雜性決定，其次由貢獻者和維護者的期望決定。

專家的期望和決策都要經過討論，但在最後，為了取得進展，必須能夠做出決策。這一特權掌握在維護人員和項目領導的手中，預計將善意使用。

因此，設定專業知識期望、作出決定和拒絕不適當的貢獻不被視為違反行為準則。

雖然維護人員一般都歡迎新來者，但他們幫助（新）貢獻者克服障礙的能力有限，因此他們必須確定優先事項。這也不應被視為違反了行為準則。內核社區意識到這一點，並以各種形式提供入門級節目，如 [kernelnewbies.org](http://kernelnewbies.org)。

## 範圍

Linux 內核社區主要在一組公共電子郵件列表上進行交互，這些列表分布在由多個不同公司或個人控制的多個不同伺服器上。所有這些列表都在內核原始碼樹中的 MAINTAINERS 文件中定義。發送到這些郵件列表的任何電子郵件都被視為包含在行為準則中。

使用 kernel.org bugzilla 和其他子系統 bugzilla 或 bug 跟蹤工具的開發人員應該遵循行為準則的指導原則。Linux 內核社區沒有「官方」項目電子郵件地址或「官方」社交媒體地址。使用 kernel.org 電子郵件帳戶執行的任何活動必須遵循為 kernel.org 發布的行為準則，就像任何使用公司電子郵件帳戶的個人必須遵循該公司的特定規則一樣。

行為準則並不禁止在郵件列表消息、內核更改日誌消息或代碼注釋中繼續包含名稱、電子郵件地址和相關注釋。

其他論壇中的互動包括在適用於上述論壇的任何規則中，通常不包括在行為準則中。除了在極端情況下可考慮的例外情況。

提交給內核的貢獻應該使用適當的語言。在行為準則之前已經存在的內容現在不會被視為違反。然而，不適當的語言可以被視為一個 bug；如果任何相關方提交補丁，這樣的 bug 將被更快地修復。當前屬於用戶/內核 API 的一部分的表達式，或者反映已發布標準或規範中使用的術語的表達式，不被視為 bug。

### 執行

行為準則中列出的地址屬於行為準則委員會。<https://kernel.org/code-of-conduct.html> 列出了在任何給定時間接收這些電子郵件的確切成員。成員不能訪問在加入委員會之前或離開委員會之後所做的報告。

最初的行为準則委員會由 TAB 的志願者以及作為中立第三方的專業調解人組成。委員會的首要任務是建立文件化的流程，並將其公開。

如果報告人不希望將整個委員會納入投訴或關切，可直接聯繫委員會的任何成員，包括調解人。

行為準則委員會根據流程審查案例（見上文），並根據需要和適當與 TAB 協商，例如請求和接收有關內核社區的信息。

委員會做出的任何決定都將提交到表中，以便在必要時與相關維護人員一起執行。行為準則委員會的決定可以通過三分之二的投票推翻。

每季度，行為準則委員會和標籤將提供一份報告，概述行為準則委員會收到的匿名報告及其狀態，以及任何否決決定的細節，包括完整和可識別的投票細節。

我們希望在啓動期之後為行為準則委員會人員配備建立一個不同的流程。發生此情況時，將使用該信息更新此文檔。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

---

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

---

### Original Documentation/process/submitting-patches.rst

譯者：

中文版維護者：鍾宇 TripleX Chung <xxx.phy@gmail.com>  
中文版翻譯者：鍾宇 TripleX Chung <xxx.phy@gmail.com>  
                  時奎亮 Alex Shi <alex.shi@linux.alibaba.com>  
中文版校譯者：李陽 Li Yang <leoyang.li@nxp.com>  
                  王聰 Wang Cong <xifyou.wangcong@gmail.com>  
                  胡皓文 Hu Haowen <src.res@email.cn>

## 如何讓你的改動進入內核

對於想要將改動提交到 Linux 內核的個人或者公司來說，如果不熟悉「規矩」，提交的流程會讓人畏懼。本文檔收集了一系列建議，這些建議可以大大的提高你的改動被接受的機會。

以下文檔含有大量簡潔的建議，具體請見：Documentation/process 同樣，[Documentation/translations/zh\\_TW/process/submit-checklist.rst](#) 紿出在提交代碼前需要檢查的項目的列表。如果你在提交一個驅動程序，那麼同時閱讀一下：Documentation/process/submitting-drivers.rst

其中許多步驟描述了 Git 版本控制系統的默認行為；如果您使用 Git 來準備補丁，您將發現它為您完成的大部分機械工作，儘管您仍然需要準備和記錄一組合理的補丁。一般來說，使用 git 將使您作為內核開發人員的生活更輕鬆。

### 0) 獲取當前源碼樹

如果您沒有一個可以使用當前內核原始碼的存儲庫，請使用 git 獲取一個。您將要從主線存儲庫開始，它可以通過以下方式獲取：

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

但是，請注意，您可能不希望直接針對主線樹進行開發。大多數子系統維護人員運行自己的樹，並希望看到針對這些樹準備的補丁。請參見 MAINTAINERS 文件中子系統的 **T:** 項以查找該樹，或者簡單地詢問維護者該樹是否未在其中列出。

仍然可以通過 tarballs 下載內核版本（如下一節所述），但這是進行內核開發的一種困難的方式。

### 1) “diff -up”

使用 “diff -up” 或者 “diff -uprN” 來創建補丁。

所有內核的改動，都是以補丁的形式呈現的，補丁由 diff(1) 生成。創建補丁的時候，要確認它是以“unified diff” 格式創建的，這種格式由 diff(1) 的 ‘-u’ 參數生成。而且，請使用 ‘-p’ 參數，那樣會顯示每個改動所在的 C 函數，使得產生的補丁容易讀得多。補丁應該基於內核原始碼樹的根目錄，而不是裡邊的任何子目錄。

為一個單獨的文件創建補丁，一般來說這樣做就夠了：

```
SRCTREE=linux
MYFILE=drivers/net/mydriver.c

cd $SRCTREE
cp $MYFILE $MYFILE.orig
vi $MYFILE      # make your change
cd ..
diff -up $SRCTREE/$MYFILE{.orig,} > /tmp/patch
```

為多個文件創建補丁，你可以解開一個沒有修改過的內核原始碼樹，然後和你自己的代碼樹之間做 diff。例如：

```
MYSRC=/devel/Linux

tar xvfz linux-3.19.tar.gz
mv linux-3.19 linux-3.19-vanilla
diff -uprN -X linux-3.19-vanilla/Documentation/dontdiff \
    linux-3.19-vanilla $MYSRC > /tmp/patch
```

“`dontdiff`”是內核在編譯的時候產生的文件的列表，列表中的文件在 `diff(1)` 產生的補丁里會被跳過。

確定你的補丁里沒有包含任何不屬於這次補丁提交的額外文件。記得在用 `diff(1)` 生成補丁之後，審閱一次補丁，以確保準確。

如果你的改動很散亂，你應該研究一下如何將補丁分割成獨立的部分，將改動分割成一系列合乎邏輯的步驟。這樣更容易讓其他內核開發者審核，如果你想你的補丁被接受，這是很重要的。請參閱：[3\) 拆分你的改動](#)

如果你用 `git`, `git rebase -i` 可以幫助你這一點。如果你不用 `git, quilt <https://savannah.nongnu.org/projects/quilt>` 另外一個流行的選擇。

## 2) 描述你的改動

描述你的問題。無論您的補丁是一行錯誤修復還是 5000 行新功能，都必須有一個潛在的問題激勵您完成這項工作。讓審稿人相信有一個問題值得解決，讓他們讀完第一段是有意義的。

描述用戶可見的影響。直接崩潰和鎖定是相當有說服力的，但並不是所有的錯誤都那麼明目張胆。即使在代碼審查期間發現了這個問題，也要描述一下您認為它可能對用戶產生的影響。請記住，大多數 Linux 安裝運行的內核來自二級穩定樹或特定於供應商/產品的樹，只從上游精選特定的補丁，因此請包含任何可以幫助您將更改定位到下游的內容：觸發的場景、`DMESG` 的摘錄、崩潰描述、性能回歸、延遲尖峯、鎖定等。

量化優化和權衡。如果您聲稱在性能、內存消耗、堆棧占用空間或二進位大小方面有所改進，請包括支持它們的數字。但也要描述不明顯的成本。優化通常不是免費的，而是在 CPU、內存和可讀性之間進行權衡；或者，探索性的工作，在不同的工作負載之間進行權衡。請描述優化的預期缺點，以便審閱者可以權衡成本和收益。

一旦問題建立起來，就要詳細地描述一下您實際在做什麼。對於審閱者來說，用簡單的英語描述代碼的變化是很重要的，以驗證代碼的行為是否符合您的意願。

如果您將補丁描述寫在一個表單中，這個表單可以很容易地作為「提交日誌」放入 Linux 的原始碼管理系統 `git` 中，那麼維護人員將非常感謝您。見[15\) 明確回覆郵件頭 \(In-Reply-To\)](#).

每個補丁只解決一個問題。如果你的描述開始變長，這就表明你可能需要拆分你的補丁。請見[3\) 拆分你的改動](#)

提交或重新提交修補程序或修補程序系列時，請包括完整的修補程序說明和理由。不要只說這是補丁（系列）的第幾版。不要期望子系統維護人員引用更早的補丁版本或引用 URL 來查找補丁描述並將其放入補丁中。也就是說，補丁（系列）及其描述應該是獨立的。這對維護人員和審查人員都有好處。一些評審者可能甚至沒有收到補丁的早期版本。

描述你在命令語氣中的變化，例如「make xyzzy do frotz」而不是「[這個補丁]make xyzzy do frotz」或「[我]changed xyzzy to do frotz」，就好像你在命令代碼庫改變它的行為一樣。

如果修補程序修復了一個記錄的 bug 條目，請按編號和 URL 引用該 bug 條目。如果補丁來自郵件列表討論，請給出郵件列表存檔的 URL；使用帶有 Message-ID 的 <https://lore.kernel.org/> 重定向，以確保連結不會過時。

但是，在沒有外部資源的情況下，儘量讓你的解釋可理解。除了提供郵件列表存檔或 bug 的 URL 之外，還要總結需要提交補丁的相關討論要點。

如果您想要引用一個特定的提交，不要只引用提交的 SHA-1 ID。還請包括提交的一行摘要，以便於審閱者了解它是關於什麼的。例如：

```
Commit e21d2170f36602ae2708 ("video: remove unnecessary
platform_set_drvdata()") removed the unnecessary
platform_set_drvdata(), but left the variable "dev" unused,
delete it.
```

您還應該確保至少使用前 12 位 SHA-1 ID。內核存儲庫包含 \* 許多 \* 對象，使與較短的 ID 發生衝突的可能性很大。記住，即使現在不會與您的六個字符 ID 發生衝突，這種情況可能五年後改變。

如果修補程序修復了特定提交中的錯誤，例如，使用 `git bisect`，請使用帶有前 12 個字符 SHA-1 ID 的“Fixes:”標記和單行摘要。為了簡化不要將標記拆分為多個，行、標記不受分析腳本「75 列換行」規則的限制。例如：

```
Fixes: 54a4f0239f2e ("KVM: MMU: make kvm_mmu_zap_page() return the number of pages it actually
   ↪ freed")
```

下列 `git config` 設置可以添加讓 `git log`, `git show` 漂亮的顯示格式：

```
[core]
    abbrev = 12
[pretty]
    fixes = Fixes: %h ("%s")
```

### 3) 拆分你的改動

將每個邏輯更改分隔成一個單獨的補丁。

例如，如果你的改動里同時有 bug 修正和性能優化，那麼把這些改動拆分到兩個或者更多的補丁文件中。如果你的改動包含對 API 的修改，並且修改了驅動程序來適應這些新的 API，那麼把這些修改分成兩個補丁。

另一方面，如果你將一個單獨的改動做成多個補丁文件，那麼將它們合併成一個單獨的補丁文件。這樣一個邏輯上單獨的改動只被包含在一個補丁文件里。

如果有一個補丁依賴另外一個補丁來完成它的改動，那沒問題。簡單的在你的補丁描述里指出「這個補丁依賴某補丁」就好了。

在將您的更改劃分為一系列補丁時，要特別注意確保內核在系列中的每個補丁之後都能正常構建和運行。使用 `git bisect` 來追蹤問題的開發者可能會在任何時候分割你的補丁系列；如果你在中間引入錯誤，他們不會感謝你。

如果你不能將補丁濃縮成更少的文件，那麼每次大約發送出 15 個，然後等待審查和集成。

### 4) 檢查你的更改風格

檢查您的補丁是否存在基本樣式衝突，詳細信息可在 [Documentation/translations/zh\\_TW/process/coding-style.rst](#) 中找到。如果不這樣做，只會浪費審稿人的時間，並且會導致你的補丁被拒絕，甚至可能沒有被閱讀。

一個重要的例外是在將代碼從一個文件移動到另一個文件時——在這種情況下，您不應該在移動代碼的同一個補丁中修改移動的代碼。這清楚地描述了移動代碼和您的更改的行為。這大大有助於審查實際差異，並允許工具更好地跟蹤代碼本身的歷史。

在提交之前，使用補丁樣式檢查程序檢查補丁 (`scripts/check patch.pl`)。不過，請注意，樣式檢查程序應該被視為一個指南，而不是作為人類判斷的替代品。如果您的代碼看起來更好，但有違規行為，那麼最好不要使用它。

檢查者報告三個級別：

- ERROR：很可能出錯的事情
- WARNING：需要仔細審查的事項
- CHECK：需要思考的事情

您應該能夠判斷您的補丁中存在的所有違規行為。

### 5) 選擇補丁收件人

您應該總是在任何補丁上複製相應的子系統維護人員，以獲得他們維護的代碼；查看維護人員文件和原始碼修訂歷史記錄，以了解這些維護人員是誰。腳本 `scripts/get_Maintainer.pl` 在這個步驟中非常有用。如果您找不到正在工作的子系統的維護人員，那麼 Andrew Morton ([akpm@linux-foundation.org](mailto:akpm@linux-foundation.org)) 將充當最後的維護人員。

您通常還應該選擇至少一個郵件列表來接收補丁集的。[linux-kernel@vger.kernel.org](mailto:linux-kernel@vger.kernel.org) 作為最後一個解決辦法的列表，但是這個列表上的體積已經引起了許多開發人員的拒絕。在 `MAINTAINERS` 文件中查找子系統特定的列表；您的補丁可能會在那裡得到更多的關注。不過，請不要發送垃圾郵件到無關的列表。

許多與內核相關的列表託管在 [vger.kernel.org](http://vger.kernel.org) 上；您可以在 <http://vger.kernel.org/vger-lists.html> 上找到它們的列表。不過，也有與內核相關的列表託管在其他地方。

不要一次發送超過 15 個補丁到 vger 郵件列表 !!!!

Linus Torvalds 是決定改動能否進入 Linux 內核的最終裁決者。他的 e-mail 地址是 <[torvalds@linux-foundation.org](mailto:torvalds@linux-foundation.org)>。他收到的 e-mail 很多，所以一般的說，最好別給他發 e-mail。

如果您有修復可利用安全漏洞的補丁，請將該補丁發送到 [security@kernel.org](mailto:security@kernel.org)。對於嚴重的 bug，可以考慮短期暫停以允許分銷商向用戶發布補丁；在這種情況下，顯然不應將補丁發送到任何公共列表。

修復已發布內核中嚴重錯誤的補丁程序應該指向穩定版維護人員，方法是放這樣的一行：

Cc: stable@vger.kernel.org

進入補丁的簽准區（注意，不是電子郵件收件人）。除了這個文件之外，您還應該閱讀 Documentation/process/stable-kernel-rules.rst

但是，請注意，一些子系統維護人員希望得出他們自己的結論，即哪些補丁應該被放到穩定的樹上。尤其是網絡維護人員，不希望看到單個開發人員在補丁中添加像上面這樣的行。

如果更改影響到用戶和內核接口，請向手冊頁維護人員（如維護人員文件中所列）發送手冊頁補丁，或至少發送更改通知，以便一些信息進入手冊頁。還應將用戶空間 API 更改複製到 [linux-api@vger.kernel.org](mailto:linux-api@vger.kernel.org)。

## 6) 沒有 MIME 編碼，沒有連結，沒有壓縮，沒有附件，只有純文本

Linus 和其他的內核開發者需要閱讀和評論你提交的改動。對於內核開發者來說，可以「引用」你的改動很重要，使用一般的 e-mail 工具，他們就可以在你的代碼的任何位置添加評論。

因為這個原因，所有的提交的補丁都是 e-mail 中「內嵌」的。

**Warning:** 如果你使用剪切-粘貼你的補丁，小心你的編輯器的自動換行功能破壞你的補丁

不要將補丁作為 MIME 編碼的附件，不管是否壓縮。很多流行的 e-mail 軟體不是任何時候都將 MIME 編碼的附件當作純文本發送的，這會使得別人無法在你的代碼中加評論。另外，MIME 編碼的附件會讓 Linus 多花一點時間來處理，這就降低了你的改動被接受的可能性。

例外：如果你的郵遞員弄壞了補丁，那麼有人可能會要求你使用 mime 重新發送補丁

請參閱 Documentation/translations/zh\_TW/process/email-clients.rst 以獲取有關配置電子郵件客戶端以使其不受影響地發送修補程序的提示。

## 7) e-mail 的大小

大的改動對郵件列表不合適，對某些維護者也不合適。如果你的補丁，在不壓縮的情況下，超過了 300kB，那麼你最好將補丁放在一個能通過 internet 訪問的服務器上，然後用指向你的補丁的 URL 替代。但是請注意，如果您的補丁超過了 300kb，那麼它幾乎肯定需要被破壞。

### 8) 回複評審意見

你的補丁幾乎肯定會得到評審者對補丁改進方法的評論。您必須對這些評論作出回應；讓補丁被忽略的一個好辦法就是忽略審閱者的意見。不會導致代碼更改的意見或問題幾乎肯定會帶來注釋或變更日誌的改變，以便下一個評審者更好地了解正在發生的事情。

一定要告訴審稿人你在做什麼改變，並感謝他們的時間。代碼審查是一個累人且耗時的過程，審查人員有時會變得暴躁。即使在這種情況下，也要禮貌地回應並解決他們指出的問題。

### 9) 不要洩氣或不耐煩

提交更改後，請耐心等待。審閱者是忙碌的人，可能無法立即訪問您的修補程序。

曾幾何時，補丁曾在沒有評論的情況下消失在空白中，但開發過程比現在更加順利。您應該在一周左右的時間內收到評論；如果沒有收到評論，請確保您已將補丁發送到正確的位置。在重新提交或聯繫審閱者之前至少等待一周-在諸如合併窗口之類的繁忙時間可能更長。

### 10) 主題中包含 PATCH

由於到 linus 和 linux 內核的電子郵件流量很高，通常會在主題行前面加上 [PATCH] 前綴。這使 Linus 和其他內核開發人員更容易將補丁與其他電子郵件討論區分開。

### 11) 簽署你的作品-開發者原始認證

為了加強對誰做了何事的追蹤，尤其是對那些透過好幾層的維護者的補丁，我們建議在發送出去的補丁上加一個「sign-off」的過程。

“sign-off”是在補丁的注釋的最後的簡單的一行文字，認證你編寫了它或者其他人有權力將它作為開放原始碼的補丁傳遞。規則很簡單：如果你能認證如下信息：

#### 開發者來源證書 1.1

對於本項目的貢獻，我認證如下信息：

- (a) 這些貢獻是完全或者部分的由我創建，我有權利以文件中指出 的開放原始碼許可證提交它；或者
- (b) 這些貢獻基於以前的工作，據我所知，這些以前的工作受恰當的開放 原始碼許可證保護，而且，根據許可證，我有權提交修改後的貢獻，無論是完全還是部分由我創造，這些貢獻都使用同一個開放原始碼許可證（除非我被允許用其它的許可證），正如文件中指出的；或者
- (c) 這些貢獻由認證 (a), (b) 或者 (c) 的人直接提供給我，而 且我沒有修改它。

(d) 我理解並同意這個項目和貢獻是公開的，貢獻的記錄（包括我一起提交的個人記錄，包括 sign-off）被永久維護並且可以和這個項目或者開放原始碼的許可證同步地再發行。

那麼加入這樣一行：

```
Signed-off-by: Random J Developer <random@developer.example.org>
```

使用你的真名（抱歉，不能使用假名或者匿名。）

有人在最後加上標籤。現在這些東西會被忽略，但是你可以這樣做，來標記公司內部的過程，或者只是指出關於 sign-off 的一些特殊細節。

如果您是子系統或分支維護人員，有時需要稍微修改收到的補丁，以便合併它們，因為樹和提交者中的代碼不完全相同。如果你嚴格遵守規則 (c)，你應該要求提交者重新發布，但這完全是在浪費時間和精力。規則 (b) 允許您調整代碼，但是更改一個提交者的代碼並讓他認可您的錯誤是非常不禮貌的。要解決此問題，建議在最後一個由簽名行和您的行之間添加一行，指示更改的性質。雖然這並不是強制性的，但似乎在描述前加上您的郵件和/或姓名（全部用方括號括起來），這足以讓人注意到您對最後一分鐘的更改負有責任。例如：

```
Signed-off-by: Random J Developer <random@developer.example.org>
[lucky@maintainer.example.org: struct foo moved from foo.c to foo.h]
Signed-off-by: Lucky K Maintainer <lucky@maintainer.example.org>
```

如果您維護一個穩定的分支機構，同時希望對作者進行致謝、跟蹤更改、合併修復並保護提交者不受投訴，那麼這種做法尤其有用。請注意，在任何情況下都不能更改作者的 ID (From 頭)，因為它是出現在更改日誌中的標識。

對回合 (back-porters) 的特別說明：在提交消息的頂部（主題行之後）插入一個補丁的起源指示似乎是一種常見且有用的實踐，以便於跟蹤。例如，下面是我們在 3.x 穩定版本中看到的內容：

```
Date: Tue Oct 7 07:26:38 2014 -0400

libata: Un-break ATA blacklist

commit 1c40279960bcd7d52dbdf1d466b20d24b99176c8 upstream.
```

還有，這裡是一個舊版內核中的一個回合補丁：

```
Date: Tue May 13 22:12:27 2008 +0200

wireless, airo: waitbusy() won't delay

[backport of 2.6 commit b7acbd9bd1f277c1eb23f344f899cf4cd0bf36a]
```

### 12) 何時使用 Acked-by: , CC: , 和 Co-Developed by:

Singed-off-by: 標記表示簽名者參與了補丁的開發，或者他/她在補丁的傳遞路徑中。

如果一個人沒有直接參與補丁的準備或處理，但希望表示並記錄他們對補丁的批准，那麼他們可以要求在補丁的變更日誌中添加一個 Acked-by:

Acked-by : 通常由受影響代碼的維護者使用，當該維護者既沒有貢獻也沒有轉發補丁時。

Acked-by: 不像簽字人那樣正式。這是一個記錄，確認人至少審查了補丁，並表示接受。因此，補丁合併有時會手動將 Acker 的「Yep, looks good to me」轉換為 Acked-By : (但請注意，通常最好要求一個明確的 Ack)。

Acked-by : 不一定表示對整個補丁的確認。例如，如果一個補丁影響多個子系統，並且有一個：來自一個子系統維護者，那麼這通常表示只確認影響維護者代碼的部分。這裡應該仔細判斷。如有疑問，應參考郵件列表檔案中的原始討論。

如果某人有機會對補丁進行評論，但沒有提供此類評論，您可以選擇在補丁中添加 Cc: 這是唯一一個標籤，它可以在沒有被它命名的人顯式操作的情況下添加，但它應該表明這個人是在補丁上抄送的。討論中包含了潛在利益相關方。

Co-developed-by: 聲明補丁是由多個開發人員共同創建的；當幾個人在一個補丁上工作時，它用於將屬性賦予共同作者（除了 From: 所賦予的作者之外）。因為 Co-developed-by: 表示作者身份，所以每個共同開發人：必須緊跟在相關合作作者的簽名之後。標準的簽核程序要求：標記的簽核順序應儘可能反映補丁的時間歷史，而不管作者是通過 From : 還是由 Co-developed-by: 共同開發的。值得注意的是，最後一個簽字人：必須始終是提交補丁的開發人員。

注意，當作者也是電子郵件標題「發件人 :」行中列出的人時，「From: 」標記是可選的。

作者提交的補丁程序示例：

```
<changelog>

Co-developed-by: First Co-Author <first@coauthor.example.org>
Signed-off-by: First Co-Author <first@coauthor.example.org>
Co-developed-by: Second Co-Author <second@coauthor.example.org>
Signed-off-by: Second Co-Author <second@coauthor.example.org>
Signed-off-by: From Author <from@author.example.org>
```

合作開發者提交的補丁示例：

```
From: From Author <from@author.example.org>

<changelog>

Co-developed-by: Random Co-Author <random@coauthor.example.org>
Signed-off-by: Random Co-Author <random@coauthor.example.org>
Signed-off-by: From Author <from@author.example.org>
Co-developed-by: Submitting Co-Author <sub@coauthor.example.org>
Signed-off-by: Submitting Co-Author <sub@coauthor.example.org>
```

### 13) 使用報告人 :: 測試人 :: 審核人 :: 建議人 :: 修復人：

**Reported-by:** 細那些發現錯誤並報告錯誤的人致謝，它希望激勵他們在將來再次幫助我們。請注意，如果 bug 是以私有方式報告的，那麼在使用 Reported-by 標記之前，請先請求權限。

**Tested-by:** 標記表示補丁已由指定的人（在某些環境中）成功測試。這個標籤通知維護人員已經執行了一些測試，為將來的補丁提供了一種定位測試人員的方法，並確保測試人員的信譽。

**Reviewed-by:** 相反，根據審查人的聲明，表明該補丁已被審查並被認為是可接受的：

#### 審查人的監督聲明

通過提供我的 Reviewed-by，我聲明：

- (a) 我已經對這個補丁進行了一次技術審查，以評估它是否適合被包含到主線內核中。
- (b) 與補丁相關的任何問題、顧慮或問題都已反饋給提交者。我對提交者對我的評論的回應感到滿意。
- (c) 雖然這一提交可能會改進一些東西，但我相信，此時，(1) 對內核進行了有價值的修改，(2) 沒有包含爭論中涉及的已知問題。
- (d) 雖然我已經審查了補丁並認為它是健全的，但我不會（除非另有明確說明）作出任何保證或保證它將在任何給定情況下實現其規定的目的或正常運行。

Reviewed-by 是一種觀點聲明，即補丁是對內核的適當修改，沒有任何遺留的嚴重技術問題。任何感興趣的審閱者（完成工作的人）都可以為一個補丁提供一個 Review-by 標籤。此標籤用於向審閱者提供致謝，並通知維護者已在修補程序上完成的審閱程度。Reviewed-by: 當由已知了解主題區域並執行徹底檢查的審閱者提供時，通常會增加補丁進入內核的可能性。

**Suggested-by:** 表示補丁的想法是由指定的人提出的，並確保將此想法歸功於指定的人。請注意，未經許可，不得添加此標籤，特別是如果該想法未在公共論壇上發布。這就是說，如果我們勤快地致謝我們的創意者，他們很有希望在未來得到鼓舞，再次幫助我們。

**Fixes:** 指示補丁在以前的提交中修復了一個問題。它可以很容易地確定錯誤的來源，這有助於檢查錯誤修復。這個標記還幫助穩定內核團隊確定應該接收修復的穩定內核版本。這是指示補丁修復的錯誤的首選方法。請參閱[2\) 描述你的改動](#) 描述您的更改以了解更多詳細信息。

### 12) 標準補丁格式

本節描述如何格式化補丁本身。請注意，如果您的補丁存儲在 Git 存儲庫中，則可以使用 `git format-patch` 進行正確的補丁格式設置。但是，這些工具無法創建必要的文本，因此請務必閱讀下面的說明。

標準的補丁，標題行是：

**Subject: [PATCH 001/123] 子系統: 一句話概述**

標準補丁的信體存在如下部分：

- 一個 “from” 行指出補丁作者。後跟空行（僅當發送修補程序的人不是作者時才需要）。
- 解釋的正文，行以 75 列包裝，這將被複製到永久變更日誌來描述這個補丁。
- 一個空行
- 上面描述的「Signed-off-by」行，也將出現在更改日誌中。
- 只包含 --- 的標記線。
- 任何其他不適合放在變更日誌的注釋。
- 實際補丁 (`diff` 輸出)。

標題行的格式，使得對標題行按字母序排序非常的容易 - 很多 e-mail 客戶端都可以支持 - 因為序列號是用零填充的，所以按數字排序和按字母排序是一樣的。

e-mail 標題中的「子系統」標識哪個內核子系統將被打補丁。

e-mail 標題中的「一句話概述」扼要的描述 e-mail 中的補丁。「一句話概述」不應該是一個文件名。對於一個補丁系列（「補丁系列」指一系列的多個相關補丁），不要對每個補丁都使用同樣的「一句話概述」。

記住 e-mail 的「一句話概述」會成為該補丁的全局唯一標識。它會蔓延到 git 的改動記錄里。然後「一句話概述」會被用在開發者的討論里，用來指代這個補丁。用戶將希望通過 google 來搜索“一句話概述”來找到那些討論這個補丁的文章。當人們在兩三個月後使用諸如 `gitk` 或 `git log --oneline` 之類的工具查看數千個補丁時，也會很快看到它。

出於這些原因，概述必須不超過 70-75 個字符，並且必須描述補丁的更改以及為什麼需要補丁。既要簡潔又要描述性很有挑戰性，但寫得好的概述應該這樣做。

概述的前綴可以用方括號括起來：「Subject: [PATCH <tag>…] <概述>」。標記不被視為概述的一部分，而是描述應該如何處理補丁。如果補丁的多個版本已發送出來以響應評審（即「v1, v2, v3」）或「rfc」，以指示評審請求，那麼通用標記可能包括版本描述符。如果一個補丁系列中有四個補丁，那麼各個補丁可以這樣編號：1/4、2/4、3/4、4/4。這可以確保開發人員了解補丁應用的順序，並且他們已經查看或應用了補丁系列中的所有補丁。

一些標題的例子：

```
Subject: [patch 2/5] ext2: improve scalability of bitmap searching
Subject: [PATCHv2 001/207] x86: fix eflags tracking
```

“From” 行是信體裡的最上面一行，具有如下格式： From: Patch Author <[author@example.com](mailto:author@example.com)>

“From” 行指明在永久改動日誌里，誰會被確認為作者。如果沒有 “From” 行，那麼郵件頭裡的 “From:” 行會被用來決定改動日誌中的作者。

說明的主題將會被提交到永久的原始碼改動日誌里，因此對那些早已經不記得和這個補丁相關的討論細節的有能力的讀者來說，是有意義的。包括補丁程序定位錯誤的（內核日誌消息、OOPS 消息等）症狀，對於搜索提交日誌以尋找適用補丁的人尤其有用。如果一個補丁修復了一個編譯失敗，那麼可能不需要包含所有編譯失敗；只要足夠讓搜索補丁的人能夠找到它就行了。與概述一樣，既要簡潔又要描述性。

“—” 標記行對於補丁處理工具要找到哪裡是改動日誌信息的結束，是不可缺少的。

對於“—”標記之後的額外註解，一個好的用途就是用來寫 diffstat，用來顯示修改了什麼文件和每個文件都增加和刪除了多少行。diffstat 對於比較大的補丁特別有用。其餘那些只是和時刻或者開發者相關的註解，不合適放到永久的改動日誌里的，也應該放這裡。使用 diffstat 的選項 “-p 1 -w 70” 這樣文件名就會從內核原始碼樹的目錄開始，不會占用太寬的空間（很容易適合 80 列的寬度，也許會有一些縮進）。

在後面的參考資料中能看到適當的補丁格式的更多細節。

## 15) 明確回覆郵件頭 (In-Reply-To)

手動添加回復補丁的標題頭 (In-Reply\_To:) 是有幫助的（例如，使用 git send-email）將補丁與以前的相關討論關聯起來，例如，將 bug 修復程序連結到電子郵件和 bug 報告。但是，對於多補丁系列，最好避免在回復時使用連結到該系列的舊版本。這樣，補丁的多個版本就不會成為電子郵件客戶端中無法管理的引用序列。如果連結有用，可以使用 <https://lore.kernel.org/> 重定向器（例如，在封面電子郵件文本中）連結到補丁系列的早期版本。

## 16) 發送 git pull 請求

如果您有一系列補丁，那麼讓維護人員通過 git pull 操作將它們直接拉入子系統存儲庫可能是最方便的。但是，請注意，從開發人員那裡獲取補丁比從郵件列表中獲取補丁需要更高的信任度。因此，許多子系統維護人員不願意接受請求，特別是來自新的未知開發人員的請求。如果有疑問，您可以在封面郵件中使用 pull 請求作為補丁系列正常發布的一個選項，讓維護人員可以選擇使用其中之一。

pull 請求的主題行中應該有 [Git Pull]。請求本身應該在一行中包含存儲庫名稱和感興趣的分支；它應該看起來像：

```
Please pull from

git://jdelvare.pck.nerim.net/jdelvare-2.6 i2c-for-linus

to get these changes:
```

pull 請求還應該包含一條整體消息，說明請求中將包含什麼，一個補丁本身的 Git shortlog 以及一個顯示補丁系列整體效果的 diffstat。當然，將所有這些信息收集在一起的最簡單方法是讓 git 使用 git request-pull 命令為您完成這些工作。

一些維護人員（包括 Linus）希望看到來自自己簽名提交的請求；這增加了他們對你的請求信心。特別是，在沒有簽名標籤的情況下，Linus 不會從像 Github 這樣的公共託管站點拉請求。

創建此類簽名的第一步是生成一個 GNRPG 密鑰，並由一個或多個核心內核開發人員對其進行簽名。這一步對新開發人員來說可能很困難，但沒有辦法繞過它。參加會議是找到可以簽署您的密鑰的開發人員的好方法。

一旦您在 Git 中準備了一個您希望有人拉的補丁系列，就用 git tag -s 創建一個簽名標記。這將創建一個新標記，標識該系列中的最後一次提交，並包含用您的私鑰創建的簽名。您還可以將 changelog 樣式的消息添加到標記中；這是一個描述拉請求整體效果的理想位置。

如果維護人員將要從中提取的樹不是您正在使用的存儲庫，請不要忘記將已簽名的標記顯式推送到公共樹。

生成拉請求時，請使用已簽名的標記作為目標。這樣的命令可以實現：

```
git request-pull master git://my.public.tree/linux.git my-signed-tag
```

### 參考文獻

**Andrew Morton, “The perfect patch” (tpp).** <<https://www.ozlabs.org/~akpm/stuff/tpp.txt>>

**Jeff Garzik, “Linux kernel patch submission format” .** <<https://web.archive.org/web/20180829112450/http://linux.yyz.us/patch-format.html>>

**Greg Kroah-Hartman, “How to piss off a kernel subsystem maintainer” .** <<http://www.kroah.com/log/linux/maintainer.html>>  
<<http://www.kroah.com/log/linux/maintainer-02.html>>  
<<http://www.kroah.com/log/linux/maintainer-03.html>>  
<<http://www.kroah.com/log/linux/maintainer-04.html>>  
<<http://www.kroah.com/log/linux/maintainer-05.html>>  
<<http://www.kroah.com/log/linux/maintainer-06.html>>

**NO!!!! No more huge patch bombs to linux-kernel@vger.kernel.org people!** <<https://lore.kernel.org/r/20050711.125305.0832243.davem@davemloft.net>>

**Kernel Documentation/process/coding-style.rst:** *Documentation/translations/zh\_TW/process/coding-style.rst*

**Linus Torvalds’ s mail on the canonical patch format:** <<https://lore.kernel.org/r/Pine.LNX.4.58.0504071023190.28951@ppc970.osdl.org>>

**Andi Kleen, “On submitting kernel patches”** Some strategies to get difficult or controversial changes in.

<http://halobates.de/on-submitting-patches.pdf>

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:src.res@email.cn)。

**Original** Documentation/process/programming-language.rst

**Translator** Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)> Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

## 程序設計語言

內核是用 C 語言 *c-language* 編寫的。更準確地說，內核通常是用 *gcc* 在 `-std=gnu11` *gcc-c-dialect-options* 下編譯的：ISO C11 的 GNU 方言

這種方言包含對語言 *gnu-extensions* 的許多擴展，當然，它們許多都在內核中使用。

對於一些體系結構，有一些使用 *clang* 和 *icc* 編譯內核的支持，儘管在編寫此文檔時還沒有完成，仍需要第三方補丁。

## 屬性

在整個內核中使用的一個常見擴展是屬性 (attributes) *gcc-attribute-syntax* 屬性允許將實現定義的語義引入語言實體（如變量、函數或類型），而無需對語言進行重大的語法更改（例如添加新關鍵字）[n2049](#)

在某些情況下，屬性是可選的（即不支持這些屬性的編譯器仍然應該生成正確的代碼，即使其速度較慢或執行的編譯時檢查/診斷次數不夠）

內核定義了偽關鍵字（例如，*pure*），而不是直接使用 GNU 屬性語法（例如，`__attribute__((__pure__))`），以檢測可以使用哪些關鍵字和/或縮短代碼，具體請參閱 `include/linux/compiler_attributes.h`

**c-language** <http://www.open-std.org/jtc1/sc22/wg14/www/standards>

**gcc** <https://gcc.gnu.org>

**clang** <https://clang.llvm.org>

**icc** <https://software.intel.com/en-us/c-compilers>

**c-dialect-options** <https://gcc.gnu.org/onlinedocs/gcc/C-Dialect-Options.html>

**gnu-extensions** <https://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html>

**gcc-attribute-syntax** <https://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html>

**n2049** <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2049.pdf>

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件

給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

### Original Documentation/process/coding-style.rst

譯者：

中文版維護者： 張樂 Zhang Le <r0bertz@gentoo.org>  
中文版翻譯者： 張樂 Zhang Le <r0bertz@gentoo.org>  
中文版校譯者： 王聰 Wang Cong <xiyou.wangcong@gmail.com>  
wheelz <kernel.zeng@gmail.com>  
管旭東 Xudong Guan <xudong.guan@gmail.com>  
Li Zefan <lizf@cn.fujitsu.com>  
Wang Chen <wangchen@cn.fujitsu.com>  
Hu Haowen <src.res@email.cn>

## Linux 內核代碼風格

這是一個簡短的文檔，描述了 linux 內核的首選代碼風格。代碼風格是因人而異的，而且我不願意把自己的觀點強加給任何人，但這就像我去做任何事情都必須遵循的原則那樣，我也希望在絕大多數事上保持這種的態度。請(在寫代碼時)至少考慮一下這裡的代碼風格。

首先，我建議你列印一份 GNU 代碼規範，然後不要讀。燒了它，這是一個具有重大象徵性意義的動作。

不管怎樣，現在我們開始：

### 1) 縮進

制表符是 8 個字符，所以縮進也是 8 個字符。有些異端運動試圖將縮進變為 4(甚至 2！)字符深，這幾乎相當於嘗試將圓周率的值定義為 3。

理由：縮進的全部意義就在於清楚的定義一個控制塊起止於何處。尤其是當你盯著你的屏幕連續看了 20 小時之後，你將會發現大一點的縮進會使你更容易分辨縮進。

現在，有些人會抱怨 8 個字符的縮進會使代碼向右邊移動的太遠，在 80 個字符的終端屏幕上就很難讀這樣的代碼。這個問題的答案是，如果你需要 3 級以上的縮進，不管用何種方式你的代碼已經有問題了，應該修正你的程序。

簡而言之，8 個字符的縮進可以讓代碼更容易閱讀，還有一個好處是當你的函數嵌套太深的時候可以給你警告。留心這個警告。

在 switch 語句中消除多級縮進的首選的方式是讓 switch 和從屬於它的 case 標籤對齊於同一列，而不要兩次縮進 case 標籤。比如：

```
switch (suffix) {
    case 'G':
    case 'g':
        mem <= 30;
        break;
```

```

case 'M':
case 'm':
    mem <= 20;
    break;
case 'K':
case 'k':
    mem <= 10;
    fallthrough;
default:
    break;
}

```

不要把多個語句放在一行里，除非你有什麼東西要隱藏：

```

if (condition) do_this;
    do_something_evertime;

```

也不要一行里放多個賦值語句。內核代碼風格超級簡單。就是避免可能導致別人誤讀的表達式。

除了注釋、文檔和 Kconfig 之外，不要使用空格來縮進，前面的例子是例外，是有意為之。

選用一個好的編輯器，不要在行尾留空格。

## 2) 把長的行和字符串打散

代碼風格的意義就在於使用平常使用的工具來維持代碼的可讀性和可維護性。

每一行的長度的限制是 80 列，我們強烈建議您遵守這個慣例。

長於 80 列的語句要打散成有意義的片段。除非超過 80 列能顯著增加可讀性，並且不會隱藏信息。子片段要明顯短於母片段，並明顯靠右。這同樣適用於有著很長參數列表的函數頭。然而，絕對不要打散對用戶可見的字符串，例如 printk 信息，因為這樣就很難對它們 grep。

## 3) 大括號和空格的放置

C 語言風格中另外一個常見問題是大括號的放置。和縮進大小不同，選擇或棄用某種放置策略並沒有多少技術上的原因，不過首選的方式，就像 Kernighan 和 Ritchie 展示給我們的，是把起始大括號放在行尾，而把結束大括號放在行首，所以：

```

if (x is true) {
    we do y
}

```

這適用於所有的非函數語句塊 (if, switch, for, while, do)。比如：

```

switch (action) {
case KOBJ_ADD:
    return "add";
}

```

```

case KOBJ_REMOVE:
    return "remove";
case KOBJ_CHANGE:
    return "change";
default:
    return NULL;
}

```

不過，有一個例外，那就是函數：函數的起始大括號放置於下一行的開頭，所以：

```

int function(int x)
{
    body of function
}

```

全世界的異端可能會抱怨這個不一致性是…呃…不一致的，不過所有思維健全的人都知道 (a) K&R 是正確的並且 (b) K&R 是正確的。此外，不管怎樣函數都是特殊的 (C 函數是不能嵌套的)。

注意結束大括號獨自占據一行，除非它後面跟著同一個語句的剩餘部分，也就是 do 語句中的“while”或者 if 語句中的“else”，像這樣：

```

do {
    body of do-loop
} while (condition);

```

和

```

if (x == y) {
    .
} else if (x > y) {
    ...
} else {
    ....
}

```

理由：K&R。

也請注意這種大括號的放置方式也能使空 (或者差不多空的) 行的數量最小化，同時不失可讀性。因此，由於你的屏幕上的新行是不可再生資源 (想想 25 行的終端屏幕)，你將會有更多的空行來放置注釋。

當只有一個單獨的語句的時候，不用加不必要的大括號。

```

if (condition)
    action();

```

和

```

if (condition)
    do_this();

```

```
else
    do_that();
```

這並不適用於只有一個條件分支是單語句的情況；這時所有分支都要使用大括號：

```
if (condition) {
    do_this();
    do_that();
} else {
    otherwise();
}
```

### 3.1) 空格

Linux 內核的空格使用方式 (主要) 取決於它是用於函數還是關鍵字。(大多數) 關鍵字後要加一個空格。值得注意的例外是 `sizeof`, `typeof`, `alignof` 和 `_attribute_`，這些關鍵字某些程度上看起來更像函數 (它們在 Linux 里也常常伴隨小括號而使用，儘管在 C 里這樣的小括號不是必需的，就像 `struct fileinfo info;` 聲明過後的 `sizeof info`)。

所以在這些關鍵字之後放一個空格：

```
if, switch, case, for, do, while
```

但是不要在 `sizeof`, `typeof`, `alignof` 或者 `_attribute_` 這些關鍵字之後放空格。例如，

```
s = sizeof(struct file);
```

不要在小括號里的表達式兩側加空格。這是一個 反例：

```
s = sizeof( struct file );
```

當聲明指針類型或者返回指針類型的函數時，\* 的首選使用方式是使之靠近變量名或者函數名，而不是靠近類型名。例子：

```
char *linux_banner;
unsigned long long memparse(char *ptr, char **retptr);
char *match_strdup(substring_t *s);
```

在大多數二元和三元操作符兩側使用一個空格，例如下面所有這些操作符：

```
= + - < > * / % | & ^ <= >= == != ? :
```

但是一元操作符後不要加空格：

```
& * + - ~ ! sizeof typeof alignof __attribute__ defined
```

後綴自加和自減一元操作符前不加空格：

```
++ --
```

前綴自加和自減一元操作符後不加空格：

```
++ --
```

. 和 -> 結構體成員操作符前後不加空格。

不要在行尾留空白。有些可以自動縮進的編輯器會在新行的行首加入適量的空白，然後你就可以直接在那一行輸入代碼。不過假如你最後沒有在那一行輸入代碼，有些編輯器就不會移除已經加入的空白，就像你故意留下一個只有空白的行。包含行尾空白的行就這樣產生了。

當 git 發現補丁包含了行尾空白的時候會警告你，並且可以應你的要求去掉行尾空白；不過如果你是正在打一系列補丁，這樣做會導致後面的補丁失敗，因為你改變了補丁的上下文。

## 4) 命名

C 是一個簡樸的語言，你的命名也應該這樣。和 Modula-2 和 Pascal 程式設計師不同，C 程式設計師不使用類似 ThisVariableIsATemporaryCounter 這樣華麗的名字。C 程式設計師會稱那個變量為 tmp，這樣寫起來會更容易，而且至少不會令其難於理解。

不過，雖然混用大小寫的名字是不提倡使用的，但是全局變量還是需要一個具描述性的名字。稱一個全局函數為 foo 是一個難以饒恕的錯誤。

全局變量（只有當你真正需要它們的時候再用它）需要有一個具描述性的名字，就像全局函數。如果你有一個可以計算活動用戶數量的函數，你應該叫它 count\_active\_users() 或者類似的名字，你不應該叫它 cntuser()。

在函數名中包含函數類型（所謂的匈牙利命名法）是腦子出了問題——編譯器知道那些類型而且能夠檢查那些類型，這樣做只能把程式設計師弄糊塗了。難怪微軟總是製造出有問題的程序。

本地變量名應該簡短，而且能夠表達相關的含義。如果你有一些隨機的整數型的循環計數器，它應該被稱為 i。叫它 loop\_counter 並無益處，如果它沒有被誤解的可能的話。類似的，tmp 可以用來稱呼任意類型的臨時變量。

如果你怕混淆了你的本地變量名，你就遇到另一個問題了，叫做函數增長荷爾蒙失衡綜合症。請看第六章（函數）。

## 5) Typedef

不要使用類似 vps\_t 之類的東西。

對結構體和指針使用 typedef 是一個錯誤。當你在代碼里看到：

```
vps_t a;
```

這代表什麼意思呢？

相反，如果是這樣

```
struct virtual_container *a;
```

你就知道 a 是什麼了。

很多人認為 `typedef` 能提高可讀性。實際不是這樣的。它們只在下列情況下有用：

- (a) 完全不透明的對象 (這種情況下要主動使用 `typedef` 來隱藏這個對象實際上是什麼)。

例如：`pte_t` 等不透明對象，你只能用合適的訪問函數來訪問它們。

**Note:** 不透明性和“訪問函數”本身是不好的。我們使用 `pte_t` 等類型的原因在於真的是完全沒有任何共用的可訪問信息。

- (b) 清楚的整數類型，如此，這層抽象就可以幫助消除到底是 `int` 還是 `long` 的混淆。

`u8/u16/u32` 是完全沒有問題的 `typedef`，不過它們更符合類別 (d) 而不是這裡。

**Note:** 要這樣做，必須事出有因。如果某個變量是 `unsigned long`，那麼沒有必要

```
typedef unsigned long myflags_t;
```

不過如果有一個明確的原因，比如它在某種情況下可能會是一個 `unsigned int` 而在其他情況下可能為 `unsigned long`，那麼就不要猶豫，請務必使用 `typedef`。

- (c) 當你使用 `sparse` 按字面的創建一個新類型來做類型檢查的時候。

- (d) 和標準 C99 類型相同的類型，在某些例外的情況下。

雖然讓眼睛和腦筋來適應新的標準類型比如 `uint32_t` 不需要花很多時間，可是有些人仍然拒絕使用它們。

因此，Linux 特有的等同於標準類型的 `u8/u16/u32/u64` 類型和它們的有符號類型是被允許的——儘管在你自己的新代碼中，它們不是強制要求要使用的。

當編輯已經使用了某個類型集的已有代碼時，你應該遵循那些代碼中已經做出的選擇。

- (e) 可以在用戶空間安全使用的類型。

在某些用戶空間可見的結構體裡，我們不能要求 C99 類型而且不能用上面提到的 `u32` 類型。因此，我們在與用戶空間共享的所有結構體中使用 `_u32` 和類似的類型。

可能還有其他的情況，不過基本的規則是 **永遠不要使用 `typedef`**，除非你可以明確的應用上述某個規則中的一個。

總的來說，如果一個指針或者一個結構體裡的元素可以合理的被直接訪問到，那麼它們就不應該是一個 `typedef`。

### 6) 函數

函數應該簡短而漂亮，並且只完成一件事情。函數應該可以一屏或者兩屏顯示完（我們都知道 ISO/ANSI 屏幕大小是 80x24），只做一件事情，而且把它做好。

一個函數的最大長度是和該函數的複雜度和縮進級數成反比的。所以，如果你有一個理論上很簡單的只有一個很長（但是簡單）的 case 語句的函數，而且你需要在每個 case 里做很多很小的事情，這樣的函數儘管很長，但也是可以的。

不過，如果你有一個複雜的函數，而且你懷疑一個天分不是很高的高中一年級學生可能甚至搞不清楚這個函數的目的，你應該嚴格遵守前面提到的長度限制。使用輔助函數，並為之取個具描述性的名字（如果你覺得它們的性能很重要的話，可以讓編譯器內聯它們，這樣的效果往往會比你寫一個複雜函數的效果要好。）

函數的另外一個衡量標準是本地變量的數量。此數量不應超過 5 – 10 個，否則你的函數就有問題了。重新考慮一下你的函數，把它分拆成更小的函數。人的大腦一般可以輕鬆的同時跟蹤 7 個不同的事物，如果再增多的話，就會糊塗了。即便你聰穎過人，你也可能會記不清你 2 個星期前做過的事情。

在源文件里，使用空行隔開不同的函數。如果該函數需要被導出，它的 **EXPORT** 宏應該緊貼在它的結束大括號之下。比如：

```
int system_is_up(void)
{
    return system_state == SYSTEM_RUNNING;
}
EXPORT_SYMBOL(system_is_up);
```

在函數原型中，包含函數名和它們的數據類型。雖然 C 語言裡沒有這樣的要求，在 Linux 里這是提倡的做法，因為這樣可以很簡單的給讀者提供更多的有價值的信息。

### 7) 集中的函數退出途徑

雖然被某些人聲稱已經過時，但是 goto 語句的等價物還是經常被編譯器所使用，具體形式是無條件跳轉指令。

當一個函數從多個位置退出，並且需要做一些類似清理的常見操作時，goto 語句就很方便了。如果並不需要清理操作，那麼直接 return 即可。

選擇一個能夠說明 goto 行為或它為何存在的標籤名。如果 goto 要釋放 buffer，一個不錯的名字可以是 `out_free_buffer:`。別去使用像 `err1:` 和 `err2:` 這樣的 GW\_BASIC 名稱，因為一旦你添加或刪除了（函數的）退出路徑，你就必須對它們重新編號，這樣會難以去檢驗正確性。

使用 goto 的理由是：

- 無條件語句容易理解和跟蹤
- 嵌套程度減小
- 可以避免由於修改時忘記更新個別的退出點而導致錯誤
- 讓編譯器省去刪除冗餘代碼的工作;)

```

int fun(int a)
{
    int result = 0;
    char *buffer;

    buffer = kmalloc(SIZE, GFP_KERNEL);
    if (!buffer)
        return -ENOMEM;

    if (condition1) {
        while (loop1) {
            ...
        }
        result = 1;
        goto out_free_buffer;
    }
    ...

out_free_buffer:
    kfree(buffer);
    return result;
}

```

一個需要注意的常見錯誤是一個 err 錯誤，就像這樣：

```

err:
    kfree(foo->bar);
    kfree(foo);
    return ret;

```

這段代碼的錯誤是，在某些退出路徑上 foo 是 NULL。通常情況下，通過把它分離成兩個錯誤標籤 err\_free\_bar: 和 err\_free\_foo: 來修復這個錯誤：

```

err_free_bar:
    kfree(foo->bar);
err_free_foo:
    kfree(foo);
    return ret;

```

理想情況下，你應該模擬錯誤來測試所有退出路徑。

## 8) 注釋

注釋是好的，不過有過度注釋的危險。永遠不要在注釋里解釋你的代碼是如何運作的：更好的做法是讓別人一看你的代碼就可以明白，解釋寫的很差的代碼是浪費時間。

一般的，你想要你的注釋告訴別人你的代碼做了什麼，而不是怎麼做的。也請你不要把注釋放在一個函數體內部：如果函數複雜到你需要獨立的注釋其中的一部分，你很可能需要回到第六章看一看。你可以做一些小注釋來註明或警告某些很聰明（或者糟糕）的做法，但不要加太多。你應該做的，是把注釋放在函數的頭部，

告訴人們它做了什麼，也可以加上它做這些事情的原因。

當注釋內核 API 函數時，請使用 kernel-doc 格式。請看 Documentation/doc-guide/ 和 scripts/kernel-doc 以獲得詳細信息。

長(多行)注釋的首選風格是：

```
/*
 * This is the preferred style for multi-line
 * comments in the Linux kernel source code.
 * Please use it consistently.
 *
 * Description: A column of asterisks on the left side,
 * with beginning and ending almost-blank lines.
 */
```

對於在 net/ 和 drivers/net/ 的文件，首選的長(多行)注釋風格有些不同。

```
/* The preferred comment style for files in net/ and drivers/net
 * looks like this.
 *
 * It is nearly the same as the generally preferred comment style,
 * but there is no initial almost-blank line.
 */
```

注釋數據也是很重要的，不管是基本類型還是衍生類型。為了方便實現這一點，每一行應只聲明一個數據(不要使用逗號來一次聲明多個數據)。這樣你就有空間來為每個數據寫一段小注釋來解釋它們的用途了。

### 9) 你已經把事情弄糟了

這沒什麼，我們都是這樣。可能你的使用了很長時間 Unix 的朋友已經告訴你 GNU emacs 能自動幫你格式化 C 原始碼，而且你也注意到了，確實是這樣，不過它所使用的默認值和我們想要的相去甚遠(實際上，甚至比隨機打的還要差——無數個猴子在 GNU emacs 里打字永遠不會創造出一個好程序)(譯註：Infinite Monkey Theorem)

所以你要麼放棄 GNU emacs，要麼改變它讓它使用更合理的設定。要採用後一個方案，你可以把下面這段粘貼到你的.emacs 文件里。

```
(defun c-lineup-arglist-tabs-only (ignored)
  "Line up argument lists by tabs, not spaces"
  (let* ((anchor (c-lang-element-pos c-syntactic-element))
         (column (c-lang-element-2nd-pos c-syntactic-element))
         (offset (- (1+ column) anchor))
         (steps (floor offset c-basic-offset)))
    (* (max steps 1)
       c-basic-offset))

(dir-locals-set-class-variables
 'linux-kernel)
```

```
'((c-mode . (
  (c-basic-offset . 8)
  (c-label-minimum-indentation . 0)
  (c-offsets-alist . (
    (arglist-close      . c-lineup-arglist-tabs-only)
    (arglist-cont-nonempty .
      (c-lineup-gcc-asm-reg c-lineup-arglist-tabs-only))
    (arglist-intro     . +)
    (brace-list-intro  . +)
    (c                  . c-lineup-C-comments)
    (case-label         . 0)
    (comment-intro     . c-lineup-comment)
    (cpp-define-intro  . +)
    (cpp-macro          . -1000)
    (cpp-macro-cont    . +)
    (defun-block-intro . +)
    (else-clause        . 0)
    (func-decl-cont    . +)
    (inclass            . +)
    (inher-cont         . c-lineup-multi-inher)
    (knr-argdecl-intro . 0)
    (label              . -1000)
    (statement          . 0)
    (statement-block-intro . +)
    (statement-case-intro . +)
    (statement-cont     . +)
    (substatement       . +)
  )))
  (indent-tabs-mode . t)
  (show-trailing-whitespace . t)
)))

(dir-locals-set-directory-class
 (expand-file-name "~/src/linux-trees")
 'linux-kernel)
```

這會讓 emacs 在 `~/src/linux-trees` 下的 C 源文件獲得更好的內核代碼風格。

不過就算你嘗試讓 emacs 正確的格式化代碼失敗了，也並不意味著你失去了一切：還可以用 `indent`。

不過，GNU `indent` 也有和 GNU emacs 一樣有問題的設定，所以你需要給它一些命令選項。不過，這還不算太糟糕，因為就算是 GNU `indent` 的作者也認同 K&R 的權威性 (GNU 的人並不是壞人，他們只是在這個問題上被嚴重的誤導了)，所以你只要給 `indent` 指定選項 `-kr -i8` (代表 K&R，8 字符縮進)，或使用 `scripts/Lindent` 這樣就可以以最時髦的方式縮進原始碼。

`indent` 有很多選項，特別是重新格式化注釋的時候，你可能需要看一下它的手冊。不過記住：`indent` 不能修正壞的編程習慣。

## 10) Kconfig 配置文件

對於遍布源碼樹的所有 Kconfig\* 配置文件來說，它們縮進方式有所不同。緊挨著 config 定義的行，用一個制表符縮進，然而 help 信息的縮進則額外增加 2 個空格。舉個例子：

```
config AUDIT
    bool "Auditing support"
    depends on NET
    help
        Enable auditing infrastructure that can be used with another
        kernel subsystem, such as SELinux (which requires this for
        logging of avc messages output). Does not do system-call
        auditing without CONFIG_AUDITSYSCALL.
```

而那些危險的功能 (比如某些文件系統的寫支持) 應該在它們的提示字符串里顯著的聲明這一點：

```
config ADFS_FS_RW
    bool "ADFS write support (DANGEROUS)"
    depends on ADFS_FS
    ...
```

要查看配置文件的完整文檔，請看 [Documentation/kbuild/kconfig-language.rst](#)。

## 11) 數據結構

如果一個數據結構，在創建和銷毀它的單線執行環境之外可見，那麼它必須要有一個引用計數器。內核里沒有垃圾收集 (並且內核之外的垃圾收集慢且效率低下)，這意味著你絕對需要記錄你對這種數據結構的使用情況。

引用計數意味著你能夠避免上鎖，並且允許多個用戶並行訪問這個數據結構——而不需要擔心這個數據結構僅僅因為暫時不被使用就消失了，那些用戶可能不過是沉睡了一陣或者做了一些其他事情而已。

注意上鎖 **不能**取代引用計數。上鎖是為了保持數據結構的一致性，而引用計數是一個內存管理技巧。通常二者都需要，不要把兩個搞混了。

很多數據結構實際上有 2 級引用計數，它們通常有不同類的用戶。子類計數器統計子類用戶的數量，每當子類計數器減至零時，全局計數器減一。

這種 多級引用計數的例子可以在內存管理 (`struct mm_struct: mm_users` 和 `mm_count`)，和文件系統 (`struct super_block: s_count` 和 `s_active`) 中找到。

記住：如果另一個執行線索可以找到你的數據結構，但這個數據結構沒有引用計數器，這裡幾乎肯定是一個 bug。

## 12) 宏，枚舉和 RTL

用於定義常量的宏的名字及枚舉里的標籤需要大寫。

```
#define CONSTANT 0x12345
```

在定義幾個相關的常量時，最好用枚舉。

宏的名字請用大寫字母，不過形如函數的宏的名字可以用小寫字母。

一般的，如果能寫成內聯函數就不要寫成像函數的宏。

含有多個語句的宏應該被包含在一個 do-while 代碼塊里：

```
#define macrofun(a, b, c)
    do {
        if (a == 5)
            do_this(b, c);
    } while (0)
```

使用宏的時候應避免的事情：

1) 影響控制流程的宏：

```
#define FOO(x)
    do {
        if (blah(x) < 0)
            return -EBUGGERED;
    } while (0)
```

非常不好。它看起來像一個函數，不過卻能導致 調用它的函數退出；不要打亂讀者大腦里的語法分析器。

2) 依賴於一個固定名字的本地變量的宏：

```
#define FOO(val) bar(index, val)
```

可能看起來像是個不錯的東西，不過它非常容易把讀代碼的人搞糊塗，而且容易導致看起來不相關的改動帶來錯誤。

3) 作為左值的帶參數的宏：FOO(x) = y；如果有人把 FOO 變成一個內聯函數的話，這種用法就會出錯了。

4) 忘記了優先級：使用表達式定義常量的宏必須將表達式置於一對小括號之內。帶參數的宏也要注意此類問題。

```
#define CONSTANT 0x4000
#define CONSTEXP (CONSTANT / 3)
```

5) 在宏里定義類似函數的本地變量時命名衝突：

```
#define FOO(x)
{
    typeof(x) ret;
    ret = calc_ret(x);
    (ret);
}
```

ret 是本地變量的通用名字 - `_foo_ret` 更不容易與一個已存在的變量衝突。

cpp 手冊對宏的講解很詳細。gcc internals 手冊也詳細講解了 RTL，內核里的彙編語言經常用到它。

### 13) 列印內核消息

內核開發者應該是受過良好教育的。請一定注意內核信息的拼寫，以給人以好的印象。不要用不規範的單詞比如 `dont`，而要用 `do not` 或者 `don't`。保證這些信息簡單明了，無歧義。

內核信息不必以英文句號結束。

在小括號里列印數字 (`%d`) 沒有任何價值，應該避免這樣做。

`<linux/device.h>` 里有一些驅動模型診斷宏，你應該使用它們，以確保信息對應於正確的設備和驅動，並且被標記了正確的消息級別。這些宏有：`dev_err()`, `dev_warn()`, `dev_info()` 等等。對於那些不和某個特定設備相關連的信息，`<linux/printk.h>` 定義了 `pr_notice()`, `pr_info()`, `pr_warn()`, `pr_err()` 和其他。

寫出好的調試信息可以是一個很大的挑戰；一旦你寫出後，這些信息在遠程除錯時能提供極大的幫助。然而列印調試信息的處理方式同列印非調試信息不同。其他 `pr_XXX()` 函數能無條件地列印，`pr_debug()` 卻不；默認情況下它不會被編譯，除非定義了 `DEBUG` 或設定了 `CONFIG_DYNAMIC_DEBUG`。實際這同樣是為了 `dev_dbg()`，一個相關約定是在一個已經開啓了 `DEBUG` 時，使用 `VERBOSE_DEBUG` 來添加 `dev_vdbg()`。

許多子系統擁有 Kconfig 調試選項來開啓 `-DDEBUG` 在對應的 Makefile 裡面；在其他情況下，特殊文件使用 `#define DEBUG`。當一條調試信息需要被無條件列印時，例如，如果已經包含一個調試相關的 `#ifdef` 條件，`printk(KERN_DEBUG ...)` 就可被使用。

### 14) 分配內存

內核提供了下面的一般用途的內存分配函數：`kmalloc()`, `kzalloc()`, `kmalloc_array()`, `kcalloc()`, `vmalloc()` 和 `vzalloc()`。請參考 API 文檔以獲取有關它們的詳細信息。

傳遞結構體大小的首選形式是這樣的：

```
p = kmalloc(sizeof(*p), ...);
```

另外一種傳遞方式中，`sizeof` 的操作數是結構體的名字，這樣會降低可讀性，並且可能會引入 bug。有可能指針變量類型被改變時，而對應的傳遞給內存分配函數的 `sizeof` 的結果不變。

強制轉換一個 void 指針返回值是多餘的。C 語言本身保證了從 void 指針到其他任何指針類型的轉換是沒有問題的。

分配一個數組的首選形式是這樣的：

```
p = kmalloc_array(n, sizeof(...), ...);
```

分配一個零長數組的首選形式是這樣的：

```
p = kcalloc(n, sizeof(...), ...);
```

兩種形式檢查分配大小  $n * \text{sizeof}(\cdots)$  的溢出，如果溢出返回 NULL。

## 15) 內聯弊病

有一個常見的誤解是 內聯是 gcc 提供的可以讓代碼運行更快的一個選項。雖然使用內聯函數有時候是恰當的（比如作為一種替代宏的方式，請看第十二章），不過很多情況下不是這樣。inline 的過度使用會使內核變大，從而使整個系統運行速度變慢。因為體積大內核會占用更多的指令高速緩存，而且會導致 pagecache 的可用內存減少。想像一下，一次 pagecache 未命中就會導致一次磁碟尋址，將耗時 5 毫秒。5 毫秒的時間內 CPU 能執行很多很多指令。

一個基本的原則是如果一個函數有 3 行以上，就不要把它變成內聯函數。這個原則的一個例外是，如果你知道某個參數是一個編譯時常量，而且因為這個常量你確定編譯器在編譯時能優化掉你的函數的大部分代碼，那仍然可以給它加上 inline 關鍵字。kmalloc() 內聯函數就是一個很好的例子。

人們經常主張給 static 的而且只用了一次的函數加上 inline，如此不會有任何損失，因為沒有什麼好權衡的。雖然從技術上說這是正確的，但是實際上這種情況下即使不加 inline gcc 也可以自動使其內聯。而且其他用戶可能會要求移除 inline，由此而來的爭論會抵消 inline 自身的潛在價值，得不償失。

## 16) 函數返回值及命名

函數可以返回多種不同類型的值，最常見的一種是表明函數執行成功或者失敗的值。這樣的一個值可以表示為一個錯誤代碼整數 (-Exxx = 失敗，0 = 成功) 或者一個 成功布爾值 (0 = 失敗，非 0 = 成功)。

混合使用這兩種表達方式是難於發現的 bug 的來源。如果 C 語言本身嚴格區分整形和布爾型變量，那麼編譯器就能夠幫我們發現這些錯誤…不過 C 語言不區分。為了避免產生這種 bug，請遵循下面的慣例：

如果函數的名字是一個動作或者強制性的命令，那麼這個函數應該返回錯誤代碼整數。如果是一個判斷，那麼函數應該返回一個 "成功" 布爾值。

比如，add work 是一個命令，所以 add\_work() 在成功時返回 0，在失敗時返回 -EBUSY。類似的，因為 PCI device present 是一個判斷，所以 pci\_dev\_present() 在成功找到一個匹配的設備時應該返回 1，如果找不到時應該返回 0。

所有 EXPORTed 函數都必須遵守這個慣例，所有的公共函數也都應該如此。私有 (static) 函數不需要如此，但是我們也推薦這樣做。

返回值是實際計算結果而不是計算是否成功的標誌的函數不受此慣例的限制。一般的，他們通過返回一些正常值範圍之外的結果來表示出錯。典型的例子是返回指針的函數，他們使用 NULL 或者 ERR\_PTR 機制來報告錯誤。

### 17) 不要重新發明內核宏

頭文件 include/linux/kernel.h 包含了一些宏，你應該使用它們，而不要自己寫一些它們的變種。比如，如果你需要計算一個數組的長度，使用這個宏

```
#define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))
```

類似的，如果你要計算某結構體成員的大小，使用

```
#define sizeof_field(t, f) (sizeof(((t*)0)->f))
```

還有可以做嚴格的類型檢查的 min() 和 max() 宏，如果你需要可以使用它們。你可以自己看看那個頭文件里還定義了什麼你可以拿來用的東西，如果有定義的話，你就不應在你的代碼里自己重新定義。

### 18) 編輯器模式行和其他需要羅嗦的事情

有一些編輯器可以解釋嵌入在源文件里的由一些特殊標記標明的配置信息。比如，emacs 能夠解釋被標記成這樣的行：

```
-*- mode: c -*-
```

或者這樣的：

```
/*
Local Variables:
compile-command: "gcc -DMAGIC_DEBUG_FLAG foo.c"
End:
*/
```

Vim 能夠解釋這樣的標記：

```
/* vim:set sw=8 noet */
```

不要在原始碼中包含任何這樣的內容。每個人都有他自己的編輯器配置，你的源文件不應該覆蓋別人的配置。這包括有關縮進和模式配置的標記。人們可以使用他們自己定製的模式，或者使用其他可以產生正確的縮進的巧妙方法。

## 19) 內聯彙編

在特定架構的代碼中，你可能需要內聯彙編與 CPU 和平台相關功能連接。需要這麼做時就不要猶豫。然而，當 C 可以完成工作時，不要平白無故地使用內聯彙編。在可能的情況下，你可以並且應該用 C 和硬體溝通。

請考慮去寫捆綁通用位元 (wrap common bits) 的內聯彙編的簡單輔助函數，別去重複地寫下只有細微差異內聯彙編。記住內聯彙編可以使用 C 參數。

大型，有一定複雜度的彙編函數應該放在.S 文件內，用相應的 C 原型定義在 C 頭文件中。彙編函數的 C 原型應該使用 `asm linkage`。

你可能需要把彙編語句標記為 `volatile`，用來阻止 GCC 在沒發現任何副作用後就把它移除了。你不必總是這樣做，儘管，這不必要的舉動會限制優化。

在寫一個包含多條指令的單個內聯彙編語句時，把每條指令用引號分割而且各占一行，除了最後一條指令外，在每個指令結尾加上 `nt`，讓彙編輸出時可以正確地縮進下一條指令：

```
asm ("magic %reg1, #42\n\t"
     "more_magic %reg2, %reg3"
     : /* outputs */ : /* inputs */ : /* clobbers */);
```

## 20) 條件編譯

只要可能，就不要在.c 文件裡面使用預處理條件 (`#if`, `#ifdef`)；這樣做讓代碼更難閱讀並且更難去跟蹤邏輯。替代方案是，在頭文件中用預處理條件提供給那些.c 文件使用，再給 `#else` 提供一個空樁 (no-op stub) 版本，然後在.c 文件內無條件地調用那些 (定義在頭文件內的) 函數。這樣做，編譯器會避免為樁函數 (stub) 的調用生成任何代碼，產生的結果是相同的，但邏輯將更加清晰。

最好傾向於編譯整個函數，而不是函數的一部分或表達式的一部分。與其放一個 `ifdef` 在表達式內，不如分解出部分或全部表達式，放進一個單獨的輔助函數，並應用預處理條件到這個輔助函數內。

如果你有一個在特定配置中，可能變成未使用的函數或變量，編譯器會警告它定義了但未使用，把它標記為 `_maybe_unused` 而不是將它包含在一個預處理條件中。(然而，如果一個函數或變量總是未使用，就直接刪除它。)

在代碼中，儘可能地使用 `IS_ENABLED` 宏來轉化某個 Kconfig 標記為 C 的布爾表達式，並在一般的 C 條件中使用它：

```
if (IS_ENABLED(CONFIG_SOMETHING)) {
    ...
}
```

編譯器會做常量摺疊，然後就像使用 `#ifdef` 那樣去包含或排除代碼塊，所以這不會帶來任何運行時開銷。然而，這種方法依舊允許 C 編譯器查看塊內的代碼，並檢查它的正確性 (語法，類型，符號引用，等等)。因此，如果條件不滿足，代碼塊內的引用符號就不存在時，你還是必須去用 `#ifdef`。

在任何有意義的 `#if` 或 `#ifdef` 塊的末尾 (超過幾行的)，在 `#endif` 同一行的後面寫下註解，注釋這個條件表達式。例如：

```
#ifdef CONFIG_SOMETHING  
...  
#endif /* CONFIG_SOMETHING */
```

### 附錄 I) 參考

The C Programming Language, 第二版作者：Brian W. Kernighan 和 Denni M. Ritchie. Prentice Hall, Inc., 1988. ISBN 0-13-110362-8 (軟皮), 0-13-110370-9 (硬皮).

The Practice of Programming 作者：Brian W. Kernighan 和 Rob Pike. Addison-Wesley, Inc., 1999. ISBN 0-201-61586-X.

GNU 手冊 - 遵循 K&R 標準和此文本 - cpp, gcc, gcc internals and indent, 都可以從 <https://www.gnu.org/manual/> 找到

WG14 是 C 語言的國際標準化工作組，URL: <http://www.open-std.org/JTC1/SC22/WG14/>

Kernel process/coding-style.rst，作者 greg@kroah.com 發表於 OLS 2002：[http://www.kroah.com/linux/talks/ols\\_2002\\_kernel\\_codingstyle\\_talk/html/](http://www.kroah.com/linux/talks/ols_2002_kernel_codingstyle_talk/html/)

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

---

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

---

**Original** Documentation/process/development-process.rst

**Translator** Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)> Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

### 內核開發過程指南

內容：

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

---

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

---

**Original** Documentation/process/1.Intro.rst

**Translator** 時奎亮 Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

校譯 吳 想 成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)> 胡 皓 文 Hu Haowen  
<[src.res@email.cn](mailto:<src.res@email.cn>)>

## 引言

### 內容提要

本節的其餘部分涵蓋了內核開發的過程，以及開發人員及其僱主在這方面可能遇到的各種問題。有很多原因使內核代碼應被合併到正式的（「主線」）內核中，包括對用戶的自動可用性、多種形式的社區支持以及影響內核開發方向的能力。提供給 Linux 內核的代碼必須在與 GPL 兼容的許可證下可用。

[開發流程如何進行](#) 介紹了開發過程、內核發布周期和合併窗口的機制。涵蓋了補丁開發、審查和合併周期中的各個階段。還有一些關於工具和郵件列表的討論？鼓勵希望開始內核開發的開發人員跟蹤並修復缺陷以作為初步練習。

[早期規劃](#) 包括項目的早期規劃，重點是儘快讓開發社區參與進來。

[使代碼正確](#) 是關於編程過程的；介紹了其他開發人員遇到的幾個陷阱。也涵蓋了對補丁的一些要求，並且介紹了一些工具，這些工具有助於確保內核補丁是正確的。

[發布補丁](#) 描述發布補丁以供評審的過程。為了讓開發社區能認真對待，補丁必須被正確格式化和描述，並且必須發送到正確的地方。遵循本節中的建議有助於確保您的工作能被較好地接納。

[跟進](#) 介紹了發布補丁之後發生的事情；工作在這時還遠遠沒有完成。與審閱者一起工作是開發過程中的一個重要部分；本節提供了一些關於如何在這個重要階段避免問題的提示。當補丁被合併到主線中時，開發人員要注意不要假定任務已經完成。

[高級主題](#) 介紹了兩個「高級」主題：使用 Git 管理補丁和查看其他人發布的補丁。

[更多信息](#) 總結了有關內核開發的更多信息，附帶有相關資源連結。

### 這個文檔是關於什麼的

Linux 內核有超過 800 萬行代碼，每個版本的貢獻者超過 1000 人，是現存最大、最活躍的免費軟體項目之一。從 1991 年開始，這個內核已經發展成為一個最好的作業系統組件，運行在袖珍數位音樂播放器、桌上型電腦、現存最大的超級計算機以及所有類型的系統上。它是一種適用於幾乎任何情況的健壯、高效和可擴展的解決方案。

隨著 Linux 的發展，希望參與其開發的開發人員（和公司）的數量也在增加。硬體供應商希望確保 Linux 能夠很好地支持他們的產品，使這些產品對 Linux 用戶具有吸引力。嵌入式系統供應商使用 Linux 作為集成產品的組件，希望 Linux 能夠儘可能地勝任手頭的任務。分銷商和其他基於 Linux 的軟體供應商切實關心 Linux 內核的功能、性能和可靠性。最終用戶也常常希望修改 Linux，使之能更好地滿足他們的需求。

Linux 最引人注目的特性之一是這些開發人員可以訪問它；任何具備必要技能的人都可以改進 Linux 並影響其開發方向。專有產品不能提供這種開放性，這是自由軟體的一個特點。如果有什麼不同的話，那就是內核比大多數其他自由軟體項目更開放。一個典型的三個月內核開發周期可以涉及 1000 多個開發人員，他們為 100 多個不同的公司（或者根本不隸屬公司）工作。

與內核開發社區合作並不是特別困難。但儘管如此，仍有許多潛在的貢獻者在嘗試做內核工作時遇到了困難。內核社區已經發展出自己獨特的操作方式，使其能夠在每天都要更改數千行代碼的環境中順利運行（並生成高質量的產品）。因此，Linux 內核開發過程與專有的開發模式有很大的不同也就不足為奇了。

對於新開發人員來說，內核的開發過程可能會讓人感到奇怪和恐懼，但這背後有充分的理由和堅實的經驗。一個不了解內核社區工作方式的開發人員（或者更糟的是，他們試圖拋棄或規避之）會得到令人沮喪的體驗。開發社區在幫助那些試圖學習的人的同時，沒有時間幫助那些不願意傾聽或不關心開發過程的人。

希望閱讀本文的人能夠避免這種令人沮喪的經歷。這些材料很長，但閱讀它們時所做的努力會在短時間內得到回報。開發社區總是需要能讓內核變更好的開發人員；下面的文字應該幫助您或為您工作的人員加入我們的社區。

### 致謝

本文檔由 Jonathan Corbet <[corbet@lwn.net](mailto:corbet@lwn.net)> 撰寫。以下人員的建議使之更為完善：Johannes Berg, James Berry, Alex Chiang, Roland Dreier, Randy Dunlap, Jake Edge, Jiri Kosina, Matt Mackall, Arthur Marsh, Amanda McPherson, Andrew Morton, Andrew Price, Tsugikazu Shibata 和 Jochen Voß。

這項工作得到了 Linux 基金會的支持，特別感謝 Amanda McPherson，他看到了這項工作的價值並將其變成現實。

## 代碼進入主線的重要性

有些公司和開發人員偶爾會想，為什麼他們要費心學習如何與內核社區合作，並將代碼放入主線內核（「主線」是由 Linus Torvalds 維護的內核，Linux 發行商將其用作基礎）。在短期內，貢獻代碼看起來像是一種可以避免的開銷；維護獨立代碼並直接支持用戶似乎更容易。事實上，保持代碼獨立（「樹外」）是在經濟上是錯誤的。

為了說明樹外代碼成本，下面給出內核開發過程的一些相關方面；本文稍後將更詳細地討論其中的大部分內容。請考慮：

- 所有 Linux 用戶都可以使用合併到主線內核中的代碼。它將自動出現在所有啓用它的發行版上。無需驅動程序磁碟、額外下載，也不需要為多個發行版的多個版本提供支持；這一切將方便所有開發人員和用戶。併入主線解決了大量的分發和支持問題。
- 當內核開發人員努力維護一個穩定的用戶空間接口時，內核內部 API 處於不斷變化之中。不維持穩定的內部接口是一個慎重的設計決策；它允許在任何時候進行基本的改進，並產出更高質量的代碼。但該策略導致結果是，若要使用新的內核，任何樹外代碼都需要持續的維護。維護樹外代碼會需要大量的工作才能使代碼保持正常運行。

相反，位於主線中的代碼不需要這樣做，因為基本規則要求進行 API 更改的任何開發人員也必須修復由於該更改而破壞的任何代碼。因此，合併到主線中的代碼大大降低了維護成本。

- 除此之外，內核中的代碼通常會被其他開發人員改進。您授權的用戶社區和客戶對您產品的改進可能會令人驚喜。
- 內核代碼在合併到主線之前和之後都要經過審查。無論原始開發人員的技能有多強，這個審查過程總是能找到改進代碼的方法。審查經常發現嚴重的錯誤和安全問題。對於在封閉環境中開發的代碼尤其如此；這種代碼從外部開發人員的審查中獲益匪淺。樹外代碼是低質量代碼。
- 參與開發過程是您影響內核開發方向的方式。旁觀者的抱怨會被聽到，但是活躍的開發人員有更強的聲音——並且能夠實現使內核更好地滿足其需求的更改。
- 當單獨維護代碼時，總是存在第三方為類似功能提供不同實現的可能性。如果發生這種情況，合併代碼將變得更加困難——甚至成為不可能。之後，您將面臨以下令人不快的選擇：(1) 無限期地維護樹外的非標準特性，或 (2) 放棄代碼並將用戶遷移到樹內版本。
- 代碼的貢獻是使整個流程工作的根本。通過貢獻代碼，您可以向內核添加新功能，並提供其他內核開發人員使用的功能和示例。如果您已經為 Linux 開發了代碼（或者正在考慮這樣做），那麼您顯然對這個平台的持續成功感興趣；貢獻代碼是確保成功的最好方法之一。

上述所有理由都適用於任何樹外內核代碼，包括以專有的、僅二進位形式分發的代碼。然而，在考慮任何類型的純二進位內核代碼分布之前，還需要考慮其他因素。包括：

- 圍繞專有內核模塊分發的法律問題其實較為模糊；相當多的內核版權所有者認為，大多數僅二進位的模塊是內核的派生產品，因此，它們的分發違反了 GNU 通用公共許可證（下面將詳細介紹）。本文作者不是律師，本文檔中的任何內容都不可能被視為法律建議。封閉原始碼模塊的真實法律地位只能由法院決定。但不管怎樣，困擾這些模塊的不確定性仍然存在。

- 二進位模塊大大增加了調試內核問題的難度，以至於大多數內核開發人員甚至都不會嘗試。因此，只分發二進位模塊將使您的用戶更難從社區獲得支持。
- 對於僅二進位的模塊的發行者來說，支持也更加困難，他們必須為他們希望支持的每個發行版和每個內核版本提供不同版本的模塊。為了提供較為全面的覆蓋範圍，可能需要一個模塊的幾十個構建，並且每次升級內核時，您的用戶都必須單獨升級這些模塊。
- 上面提到的關於代碼評審的所有問題都更加存在於封閉原始碼中。由於該代碼根本不可得，因此社區無法對其進行審查，毫無疑問，它將存在嚴重問題。

尤其是嵌入式系統的製造商，可能會傾向於忽視本節中所說的大部分內容；因為他們相信自己正在商用一種使用凍結內核版本的獨立產品，在發布後不需要再進行開發。這個論點忽略了廣泛的代碼審查的價值以及允許用戶向產品添加功能的價值。但這些產品的商業壽命有限，之後必須發布新版本的產品。在這一點上，代碼在主線上並得到良好維護的供應商將能夠更好地占位，以使新產品快速上市。

## 許可

代碼是根據一些許可證提供給 Linux 內核的，但是所有代碼都必須與 GNU 通用公共許可證（GPLv2）的版本 2 兼容，該版本是覆蓋整個內核分發的許可證。在實踐中，這意味著所有代碼貢獻都由 GPLv2（可選地，語言允許在更高版本的 GPL 下分發）或 3 子句 BSD 許可（New BSD License，譯者注）覆蓋。任何不包含在兼容許可證中的貢獻都不會被接受到內核中。

貢獻給內核的代碼不需要（或請求）版權分配。合併到主線內核中的所有代碼都保留其原始所有權；因此，內核現在擁有數千個所有者。

這種所有權結構也暗示著，任何改變內核許可的嘗試都註定會失敗。很少有實際情況可以獲得所有版權所有者的同意（或者從內核中刪除他們的代碼）。因此，尤其是在可預見的將來，許可證不大可能遷移到 GPL 的版本 3。

所有貢獻給內核的代碼都必須是合法的免費軟體。因此，不接受匿名（或化名）貢獻者的代碼。所有貢獻者都需要在他們的代碼上「sign off（簽發）」，聲明代碼可以在 GPL 下與內核一起分發。無法提供未被其所有者許可為免費軟體的代碼，或可能為內核造成版權相關問題的代碼（例如，由缺乏適當保護的反向工程工作派生的代碼）不能被接受。

有關版權問題的提問在 Linux 開發郵件列表中很常見。這樣的問題通常會得到不少答案，但請記住，回答這些問題的人不是律師，不能提供法律諮詢。如果您有關於 Linux 原始碼的法律問題，沒有什麼可以代替諮詢了解這一領域的律師。依賴從技術郵件列表中獲得的答案是一件冒險的事情。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

---

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件

給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

**Original** Documentation/process/2.Process.rst

**Translator** 時奎亮 Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

校譯 吳 想 成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)> 胡 皓 文 Hu Haowen  
<[src.res@email.cn](mailto:<src.res@email.cn>)>

## 開發流程如何進行

90 年代早期的 Linux 內核開發是一件相當鬆散的事情，涉及的用戶和開發人員相對較少。由於擁有數以百萬計的用戶羣，且每年有大約 2000 名開發人員參與進來，內核因此必須發展出許多既定流程來保證開發的順利進行。要參與到流程中來，需要對此流程的進行方式有一個紮實的理解。

### 總覽

內核開發人員使用一個鬆散的基於時間的發布過程，每兩到三個月發布一次新的主要內核版本。最近的發布歷史記錄如下：

5.0	2019 年 3 月 3 日
5.1	2019 年 5 月 5 日
5.2	2019 年 7 月 7 日
5.3	2019 年 9 月 15 日
5.4	2019 年 11 月 24 日
5.5	2020 年 1 月 6 日

每個 5.x 版本都是一個主要的內核版本，具有新特性、內部 API 更改等等。一個典型的 5.x 版本包含大約 13000 個變更集，變更了幾十萬行代碼。因此，5.x 是 Linux 內核開發的前沿；內核使用滾動開發模型，不斷集成重大變化。

對於每個版本的補丁合併，遵循一個相對簡單的規則。在每個開發周期的開頭，「合併窗口」被打開。這時，被認為足夠穩定（並且被開發社區接受）的代碼被合併到主線內核中。在這段時間內，新開發周期的大部分變更（以及所有主要變更）將以接近每天 1000 次變更（「補丁」或「變更集」）的速度合併。

（順便說一句，值得注意的是，合併窗口期間集成的更改並不是憑空產生的；它們是經提前收集、測試和分級的。稍後將詳細描述該過程的工作方式。）

合併窗口持續大約兩周。在這段時間結束時，Linus Torvalds 將聲明窗口已關閉，並釋放第一個「rc」內核。例如，對於目標為 5.6 的內核，在合併窗口結束時發生的釋放將被稱為 5.6-rc1。-rc1 版本是一個信號，表示合併新特性的時間已經過去，穩定下一個內核的時間已經到來。

在接下來的 6 到 10 周內，只有修復問題的補丁才應該提交給主線。有時會允許更大的更改，但這種情況很少發生；試圖在合併窗口外合併新功能的開發人員往往受不到友好的接待。一般來說，如果您錯過了給定特

性的合併窗口，最好的做法是等待下一個開發周期。（偶爾會對未支持硬體的驅動程序進行例外；如果它們不改變已有代碼，則不會導致回歸，應該可以隨時被安全地加入）。

隨著修復程序進入主線，補丁速度將隨著時間的推移而變慢。Linus 大約每周發布一次新的-rc 內核；在內核被認為足夠穩定並最終發布前，一般會達到-rc6 到-rc9 之間。然後，整個過程又重新開始了。

例如，這裡是 5.4 的開發周期進行情況（2019 年）：

九月 15	5.3 穩定版發布
九月 30	5.4-rc1 合併窗口關閉
十月 6	5.4-rc2
十月 13	5.4-rc3
十月 20	5.4-rc4
十月 27	5.4-rc5
十一月 3	5.4-rc6
十一月 10	5.4-rc7
十一月 17	5.4-rc8
十一月 24	5.4 穩定版發布

開發人員如何決定何時結束開發周期並創建穩定版本？最重要的指標是以前版本的回歸列表。不歡迎出現任何錯誤，但是那些破壞了以前能工作的系統的錯誤被認為是特別嚴重的。因此，導致回歸的補丁是不受歡迎的，很可能在穩定期內刪除。

開發人員的目標是在穩定發布之前修復所有已知的回歸。在現實世界中，這種完美是很難實現的；在這種規模的項目中，變數太多了。需要說明的是，延遲最終版本只會使問題變得更糟；等待下一個合併窗口的更改將變多，導致下次出現更多的回歸錯誤。因此，大多數 5.x 內核都有一些已知的回歸錯誤，不過，希望沒有一個是嚴重的。

一旦一個穩定的版本發布，它的持續維護工作就被移交給「穩定團隊」，目前由 Greg Kroah-Hartman 領導。穩定團隊將使用 5.x.y 編號方案不定期地發布穩定版本的更新。要合入更新版本，補丁必須（1）修復一個重要的缺陷，且（2）已經合併到下一個開發版本主線中。內核通常會在其初始版本後的一個以上的開發周期內收到穩定版更新。例如，5.2 內核的歷史如下（2019 年）：

七月 7	5.2 穩定版發布
七月 13	5.2.1
七月 21	5.2.2
七月 26	5.2.3
七月 28	5.2.4
七月 31	5.2.5
...	...
十月 11	5.2.21

5.2.21 是 5.2 版本的最終穩定更新。

有些內核被指定為「長期」內核；它們將得到更長時間的支持。在本文中，當前的長期內核及其維護者是：

3.16	Ben Hutchings	(長期穩定內核)
4.4	Greg Kroah-Hartman & Sasha Levin	(長期穩定內核)
4.9	Greg Kroah-Hartman & Sasha Levin	
4.14	Greg Kroah-Hartman & Sasha Levin	
4.19	Greg Kroah-Hartman & Sasha Levin	
5.4	Greg Kroah-Hartman & Sasha Levin	

長期支持內核的選擇純粹是維護人員是否有需求和時間來維護該版本的問題。目前還沒有為即將發布的任何特定版本提供長期支持的已知計劃。

## 補丁的生命周期

補丁不會直接從開發人員的鍵盤進入主線內核。相反，有一個稍微複雜（如果有些非正式）的過程，旨在確保對每個補丁進行質量審查，並確保每個補丁實現了一個在主線中需要的更改。對於小的修復，這個過程可能會很快完成，而對於較大或有爭議的變更，可能會持續數年。許多開發人員的沮喪來自於對這個過程缺乏理解或者試圖繞過它。

為了減少這種挫敗，本文將描述補丁如何進入內核。下面的介紹以一種較為理想化的方式描述了這個過程。更詳細的過程將在後面的章節中介紹。

補丁通常要經歷以下階段：

- 設計。這就是補丁的真正需求——以及滿足這些需求的方式——所在。設計工作通常是在不涉及社區的情況下完成的，但是如果可能的話，最好是在公開的情況下完成這項工作；這樣可以節省很多稍後再重新設計的時間。
- 早期評審。補丁被發布到相關的郵件列表中，列表中的開發人員會回復他們可能有的任何評論。如果一切順利的話，這個過程應該會發現補丁的任何主要問題。
- 更廣泛的評審。當補丁接近準備好納入主線時，它應該被相關的子系統維護人員接受——儘管這種接受並不能保證補丁會一直延伸到主線。補丁將出現在維護人員的子系統樹中，並進入 -next 樹（如下所述）。當流程進行時，此步驟將會對補丁進行更廣泛的審查，並發現由於將此補丁與其他人所做的工作合併而導致的任何問題。
- 請注意，大多數維護人員也有日常工作，因此合併補丁可能不是他們的最優先工作。如果您的補丁得到了需要更改的反饋，那麼您應該進行這些更改，或者解釋為何不應該進行這些更改。如果您的補丁沒有評審意見，也沒有被其相應的子系統或驅動程序維護者接受，那麼您應該堅持不懈地將補丁更新到當前內核使其可被正常應用，並不斷地發送它以供審查和合併。
- 合併到主線。最終，一個成功的補丁將被合併到由 Linus Torvalds 管理的主線存儲庫中。此時可能會出現更多的評論和/或問題；對開發人員來說應對這些問題並解決出現的任何問題仍很重要。
- 穩定版發布。大量用戶可能受此補丁影響，因此可能再次出現新的問題。
- 長期維護。雖然開發人員在合併代碼後可能會忘記代碼，但這種行為往往會給開發社區留下不良印象。合併代碼消除了一些維護負擔，因為其他人將修復由 API 更改引起的問題。但是，如果代碼要長期保持

可用，原始開發人員應該繼續為代碼負責。

內核開發人員（或他們的僱主）犯的最大錯誤之一是試圖將流程簡化為一個「合併到主線」步驟。這種方法總是會讓所有相關人員感到沮喪。

### 補丁如何進入內核

只有一個人可以將補丁合併到主線內核存儲庫中：Linus Torvalds。但是，在進入 2.6.38 內核的 9500 多個補丁中，只有 112 個（大約 1.3%）是由 Linus 自己直接選擇的。內核項目已經發展到一個沒有一個開發人員可以在沒有支持的情況下檢查和選擇每個補丁的規模。內核開發人員處理這種增長的方式是使用圍繞信任鏈構建的助理系統。

內核代碼庫在邏輯上被分解為一組子系統：網絡、特定體系結構支持、內存管理、視頻設備等。大多數子系統都有一個指定的維護人員，其總體負責該子系統中的代碼。這些子系統維護者（鬆散地）是他們所管理的內核部分的「守門員」；他們（通常）會接受一個補丁以包含到主線內核中。

子系統維護人員每個人都管理著自己版本的內核原始碼樹，通常（並非總是）使用 Git。Git 等工具（以及 Quilt 或 Mercurial 等相關工具）允許維護人員跟蹤補丁列表，包括作者信息和其他元數據。在任何給定的時間，維護人員都可以確定他或她的存儲庫中的哪些補丁在主線中找不到。

當合併窗口打開時，頂級維護人員將要求 Linus 從存儲庫中「拉出」他們為合併選擇的補丁。如果 Linus 同意，補丁流將流向他的存儲庫，成為主線內核的一部分。Linus 對拉取中接收到的特定補丁的關注程度各不相同。很明顯，有時他看起來很關注。但是一般來說，Linus 相信子系統維護人員不會向上游發送壞補丁。

子系統維護人員反過來也可以從其他維護人員那裡獲取補丁。例如，網絡樹是由首先在專用於網絡設備驅動程序、無線網絡等的樹中積累的補丁構建的。此存儲鏈可以任意長，但很少超過兩個或三個連結。由於鏈中的每個維護者都信任那些管理較低級別樹的維護者，所以這個過程稱為「信任鏈」。

顯然，在這樣的系統中，獲取內核補丁取決於找到正確的維護者。直接向 Linus 發送補丁通常不是正確的方法。

### Next 樹

子系統樹鏈引導補丁流到內核，但它也提出了一個有趣的問題：如果有人想查看為下一個合併窗口準備的所有補丁怎麼辦？開發人員將感興趣的是，還有什麼其他的更改有待解決，以了解是否存在需要擔心的衝突；例如，更改核心內核函數原型的修補程序將與使用該函數舊形式的任何其他修補程序衝突。審查人員和測試人員希望在所有這些變更到達主線內核之前，能夠訪問它們的集成形式的變更。您可以從所有相關的子系統樹中提取更改，但這將是一項複雜且容易出錯的工作。

解決方案以-next 樹的形式出現，在這裡子系統樹被收集以供測試和審查。這些樹中由 Andrew Morton 維護的較老的一個，被稱為「-mm」（用於內存管理，創建時為此）。-mm 樹集成了一長串子系統樹中的補丁；它還包含一些旨在幫助調試的補丁。

除此之外，-mm 還包含大量由 Andrew 直接選擇的補丁。這些補丁可能已經發布在郵件列表上，或者它們可能應用於內核中未指定子系統樹的部分。同時，-mm 作為最後手段的子系統樹；如果沒有其他明顯的路徑

可以讓補丁進入主線，那麼它很可能最終選擇-mm 樹。累積在-mm 中的各種補丁最終將被轉發到適當的子系統樹，或者直接發送到 Linus。在典型的開發周期中，大約 5-10% 的補丁通過-mm 進入主線。

當前-mm 補丁可在「mmotm」(-mm of the moment) 目錄中找到：

<https://www.ozlabs.org/~akpm/mmotm/>

然而，使用 MMOTM 樹可能會十分令人頭疼；它甚至可能無法編譯。

下一個周期補丁合併的主要樹是 linux-next，由 Stephen Rothwell 維護。根據設計 linux-next 是下一個合併窗口關閉後主線的快照。linux-next 樹在 Linux-kernel 和 Linux-next 電郵列表中發布，可從以下位置下載：

<https://www.kernel.org/pub/linux/kernel/next/>

Linux-next 已經成為內核開發過程中不可或缺的一部分；在一個給定的合併窗口中合併的所有補丁都應該在合併窗口打開之前的一段時間內找到進入 Linux-next 的方法。

## Staging 樹

內核原始碼樹包含 drivers/staging/目錄，其中有許多驅動程序或文件系統的子目錄正在被添加到內核樹中。它們在仍然需要更多的修正的時候可以保留在 driver/staging/ 目錄中；一旦完成，就可以將它們移到內核中。這是一種跟蹤不符合 Linux 內核編碼或質量標準的驅動程序的方法，人們可能希望使用它們並跟蹤開發。

Greg Kroah Hartman 目前負責維護 staging 樹。仍需要修正的驅動程序將發送給他，每個驅動程序在 drivers/staging/ 中都有自己的子目錄。除了驅動程序源文件之外，目錄中還應該有一個 TODO 文件。TODO 文件列出了驅動程序需要接受的暫停的工作，以及驅動程序的任何補丁都應該抄送的人員列表。當前的規則要求，staging 的驅動程序必須至少正確編譯。

Staging 是一種讓新的驅動程序進入主線的相對容易的方法，它們會幸運地引起其他開發人員的注意，並迅速改進。然而，進入 staging 並不是故事的結尾；staging 中沒有看到常規進展的代碼最終將被刪除。經銷商也傾向於相對不願意使用 staging 驅動程序。因此，在成為一個合適的主線驅動的路上，staging 僅是一個中轉站。

## 工具

從上面的文本可以看出，內核開發過程在很大程度上依賴於在不同方向上聚集補丁的能力。如果沒有適當強大的工具，整個系統將無法在任何地方正常工作。關於如何使用這些工具的教程遠遠超出了本文檔的範圍，但還是用一點篇幅介紹一些關鍵點。

到目前為止，內核社區使用的主要原始碼管理系統是 git。Git 是在自由軟體社區中開發的許多分布式版本控制系統之一。它非常適合內核開發，因為它在處理大型存儲庫和大量補丁時性能非常好。它也以難以學習和使用而著稱，儘管隨著時間的推移它變得更好了。對於內核開發人員來說，對 Git 的某種熟悉幾乎是一種要求；即使他們不將它用於自己的工作，他們也需要 Git 來跟上其他開發人員（以及主線）正在做的事情。

現在幾乎所有的 Linux 發行版都打包了 Git。Git 主頁位於：

<https://git-scm.com/>

此頁面包含了文檔和教程的連結。

在不使用 git 的內核開發人員中，最流行的選擇幾乎肯定是 Mercurial：

<http://www.seleric.com/mercurial/>

Mercurial 與 Git 共享許多特性，但它提供了一個界面，許多人覺得它更易於使用。

另一個值得了解的工具是 Quilt：

<https://savannah.nongnu.org/projects/quilt>

Quilt 是一個補丁管理系統，而不是原始碼管理系統。它不會隨著時間的推移跟蹤歷史；相反，它面向根據不斷發展的代碼庫跟蹤一組特定的更改。一些主要的子系統維護人員使用 Quilt 來管理打算向上游移動的補丁。對於某些樹的管理（例如-mm），quilt 是最好的工具。

## 郵件列表

大量的 Linux 內核開發工作是通過郵件列表完成的。如果不加入至少一個某個列表，就很難成為社區中的一個「全功能」成員。但是，Linux 郵件列表對開發人員來說也是一個潛在的危險，他們可能會被一堆電子郵件淹沒、違反 Linux 列表上使用的約定，或者兩者兼而有之。

大多數內核郵件列表都在 vger.kernel.org 上運行；主列表位於：

<http://vger.kernel.org/vger-lists.html>

不過，也有一些列表託管在別處；其中一些列表位於 redhat.com/mailman/listinfo。

當然，內核開發的核心郵件列表是 linux-kernel。這個列表是一個令人生畏的地方：每天的信息量可以達到 500 條，噪音很高，談話技術性很強，且參與者並不總是表現出高度的禮貌。但是，沒有其他地方可以讓內核開發社區作為一個整體聚集在一起；不使用此列表的開發人員將錯過重要信息。

以下一些提示可以幫助在 linux-kernel 生存：

- 將郵件轉移到單獨的文件夾，而不是主郵箱文件夾。我們必須能夠持續地忽略洪流。
- 不要試圖跟上每一次談話——沒人會這樣。重要的是要篩選感興趣的主題（但請注意長時間的對話可能會偏離原來的主題，儘管未改變電子郵件的主題）和參與的人。
- 不要回復挑事的人。如果有人試圖激起憤怒，請忽略他們。
- 當回復 Linux 內核電子郵件（或其他列表上的電子郵件）時，請為所有相關人員保留 Cc: 抄送頭。如果沒有確實的理由（如明確的請求），則不應刪除收件人。一定要確保你要回復的人在抄送列表中。這個慣例也使你不必在回覆郵件時明確要求被抄送。
- 在提出問題之前，搜索列表存檔（和整個網絡）。有些開發人員可能會對那些顯然沒有完成家庭作業的人感到不耐煩。
- 避免頂部回復（把你的答案放在你要回復的引文上面的做法）。這會讓你的回答更難理解，印象也很差。

- 在正確的郵件列表發問。linux-kernel 可能是通用的討論場所，但它不是尋找所有子系統開發人員的最佳場所。

最後一點——找到正確的郵件列表——是開發人員常出錯的地方。在 linux-kernel 上提出與網絡相關的問題的人幾乎肯定會收到一個禮貌的建議，轉到 netdev 列表上提出，因為這是大多數網絡開發人員經常出現的列表。還有其他列表可用於 scsi、video4linux、ide、filesystem 等子系統。查找郵件列表的最佳位置是與內核原始碼一起打包的 MAINTAINERS 文件。

## 開始內核開發

關於如何開始內核開發過程的問題很常見——個人和公司皆然。同樣常見的是失誤，這使得關係的開始比本應的更困難。

公司通常希望聘請知名的開發人員來啓動開發團隊。實際上，這是一種有效的技術。但它也往往是昂貴的，而且對增加有經驗的內核開發人員的數量沒有多大幫助。考慮到時間投入，可以讓內部開發人員加快 Linux 內核的開發速度。利用這段時間可以讓僱主擁有一批既了解內核又了解公司的開發人員，還可以幫助培訓其他人。從中期來看，這通常是更有利可圖的方法。

可以理解的是，單個開發人員往往對起步感到茫然。從一個大型項目開始可能會很嚇人；人們往往想先用一些較小的東西來試試水。由此，一些開發人員開始創建修補拼寫錯誤或輕微編碼風格問題的補丁。不幸的是，這樣的補丁會產生一定程度的噪音，這會分散整個開發社區的注意力，因此，它們越來越被人不看重。希望向社區介紹自己的新開發人員將無法通過這些方式獲得他們期待的反響。

Andrew Morton 為有抱負的內核開發人員提供了如下建議

所有內核開發者的第一個項目肯定應該是「確保內核在您可以操作的所有機器上始終完美運行」。通常的方法是和其他人一起解決問題（這可能需要堅持！），但就是如此——這是內核開發的一部分。

(<http://lwn.net/articles/283982/>)

在沒有明顯問題需要解決的情況下，通常建議開發人員查看當前的回歸和開放缺陷列表。從來都不缺少需要解決的問題；通過解決這些問題，開發人員將從該過程獲得經驗，同時與開發社區的其他成員建立相互尊重。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

---

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

**Original** Documentation/process/3.Early-stage.rst

**Translator** 時奎亮 Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

校譯 吳 想 成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)> 胡 皓 文 Hu Haowen  
<[src.res@email.cn](mailto:src.res@email.cn)>

### 早期規劃

當考慮一個 Linux 內核開發項目時，很可能會直接跳進去開始編碼。然而，與任何重要的項目一樣，許多成功的基礎最好是在第一行代碼編寫之前就打下。在早期計劃和溝通中花費一些時間可以在之後節省更多的時間。

### 搞清問題

與任何工程項目一樣，成功的內核改善從清晰描述要解決的問題開始。在某些情況下，這個步驟很容易：例如當某個特定硬體需要驅動程序時。不過，在其他情況下，很容易將實際問題與建議的解決方案混在一起，這可能會導致麻煩。

舉個例子：幾年前，Linux 音頻的開發人員尋求一種方法來運行應用程式，而不會因系統延遲過大而導致退出或其他問題。他們得到的解決方案是一個連接到 Linux 安全模塊（LSM）框架中的內核模塊；這個模塊可以配置為允許特定的應用程式訪問實時調度程序。這個模塊被實現並發到 `linux-kernel` 電子郵件列表，在那裡它立即遇到了麻煩。

對於音頻開發人員來說，這個安全模塊足以解決他們當前的問題。但是，對於更廣泛的內核社區來說，這被視為對 LSM 框架的濫用（LSM 框架並不打算授予他們原本不具備的進程特權），並對系統穩定性造成風險。他們首選的解決方案包括短期的通過 `rlimit` 機制進行實時調度訪問，以及長期的減少延遲的工作。

然而，音頻社區無法超越他們實施的特定解決方案來看問題；他們不願意接受替代方案。由此產生的分歧使這些開發人員對整個內核開發過程感到失望；其中一個開發人員返回到 `audio` 列表並發布了以下內容：

有很多非常好的 Linux 內核開發人員，但他們往往會被一羣傲慢的傻瓜所壓倒。試圖向這些人傳達用戶需求是浪費時間。他們太「聰明」了，根本聽不到少數人的話。

(<http://lwn.net/articles/131776/>)

實際情況卻是不同的；與特定模塊相比，內核開發人員更關心系統穩定性、長期維護以及找到問題的正確解決方案。這個故事的寓意是把重點放在問題上——而不是具體的解決方案上——並在開始編寫代碼之前與開發社區討論這個問題。

因此，在考慮一個內核開發項目時，我們應該得到一組簡短問題的答案：

- 需要解決的問題究竟是什麼？
- 受此問題影響的用戶有哪些？解決方案應該解決哪些使用案例？
- 內核現在為何沒能解決這個問題？

只有這樣，才能開始考慮可能的解決方案。

## 早期討論

在計劃內核開發項目時，在開始實施之前與社區進行討論是很有意義的。早期溝通可以通過多種方式節省時間和麻煩：

- 很可能問題是由內核以您不理解的方式解決的。Linux 內核很大，具有許多不明顯的特性和功能。並不是所有的內核功能都像人們所希望的那樣有文檔記錄，而且很容易遺漏一些東西。某作者發布了一個完整的驅動程序，重複了一個其不知道的現有驅動程序。重新發明現有輪子的代碼不僅浪費，而且不會被接受到主線內核中。
- 建議的解決方案中可能有一些要素不適合併入主線。在編寫代碼之前，最好先了解這樣的問題。
- 其他開發人員完全有可能考慮過這個問題；他們可能有更好的解決方案的想法，並且可能願意幫助創建這個解決方案。

在內核開發社區的多年經驗給了我們一個明確的教訓：閉門設計和開發的內核代碼總是有一些問題，這些問題只有在代碼發布到社區中時才會被發現。有時這些問題很嚴重，需要數月或數年的努力才能使代碼達到內核社區的標準。例如：

- 設計並實現了單處理器系統的 DeviceScape 網絡棧。只有使其適合於多處理器系統，才能將其併到主線中。在代碼中修改鎖等等是一項困難的任務；因此，這段代碼（現在稱為 mac80211）的併被推遲了一年多。
- Reiser4 文件系統包含許多功能，核心內核開發人員認為這些功能應該在虛擬文件系統層中實現。它還包括一些特性，這些特性在不將系統暴露於用戶引起的死鎖的情況下是不容易實現的。這些問題過遲發現——以及拒絕處理其中一些問題——已經導致 Reiser4 置身主線內核之外。
- Apparmor 安全模塊以被認為不安全和不可靠的方式使用內部虛擬文件系統數據結構。這種擔心（包括其他）使 Apparmor 多年來無法進入主線。

在這些情況下，與內核開發人員的早期討論，可以避免大量的痛苦和額外的工作。

## 找誰交流？

當開發人員決定公開他們的計劃時，下一個問題是：我們從哪裡開始？答案是找到正確的郵件列表和正確的維護者。對於郵件列表，最好的方法是在維護者 (MAINTAINERS) 文件中查找要發布的相關位置。如果有一個合適的子系統列表，那麼其上發布通常比在 linux-kernel 上發布更可取；您更有可能接觸到在相關子系統中具有專業知識的開發人員，並且環境可能具支持性。

找到維護人員可能會有點困難。同樣，維護者文件是開始的地方。但是，該文件往往不是最新的，並且並非所有子系統都在那裡顯示。實際上，維護者文件中列出的人員可能不是當前實際擔任該角色的人員。因此，當對聯繫誰有疑問時，一個有用的技巧是使用 git（尤其是「git-log」）查看感興趣的子系統中當前活動的用戶。看看誰在寫補丁、誰會在這些補丁上加上 Signed-off-by 行簽名（如有）。這些人將是幫助新開發項目的最佳人選。

找到合適的維護者有時是非常具有挑戰性的，以至於內核開發人員添加了一個腳本來簡化這個過程：

.../scripts/get\_maintainer.pl

當給定「-f」選項時，此腳本將返回指定文件或目錄的當前維護者。如果在命令行上給出了一個補丁，它將列出可能接收補丁副本的維護人員。有許多選項可以調節 `get_maintainer.pl` 搜索維護者的嚴格程度；請小心使用更激進的選項，因為最終結果可能會包括對您正在修改的代碼沒有真正興趣的開發人員。

如果所有其他方法都失敗了，那麼與 Andrew Morton 交流是跟蹤特定代碼段維護人員的一種有效方法。

### 何時郵寄？

如果可能的話，在早期階段發布你的計劃只會更有幫助。描述正在解決的問題以及已經制定的關於如何實施的任何計劃。您可以提供的任何信息都可以幫助開發社區為項目提供有用的輸入。

在這個階段可能發生的一件令人沮喪的事情不是得到反對意見，而是很少或根本沒有反饋。令人傷心的事實是：(1) 內核開發人員往往很忙；(2) 不缺少有宏偉計劃但代碼（甚至代碼設想）很少的人去支持他們；(3) 沒有人有義務審查或評論別人發表的想法。除此之外，高層級的設計常常隱藏著一些問題，這些問題只有在有人真正嘗試實現這些設計時才會被發現；因此，內核開發人員寧願看到代碼。

如果發布請求評論（RFC）並沒得到什麼有用的評論，不要以為這意味著無人對此項目有興趣，同時你也不能假設你的想法沒有問題。在這種情況下，最好的做法是繼續進行，把你的進展隨時通知社區。

### 獲得官方認可

如果您的工作是在公司環境中完成的，就像大多數 Linux 內核工作一樣；顯然，在您將公司的計劃或代碼發布到公共郵件列表之前，必須獲得有適當權利經理的許可。發布不確定是否兼容 GPL 的代碼尤其會帶來問題；公司的管理層和法律人員越早能夠就發布內核開發項目達成一致，對參與的每個人都越好。

一些讀者可能會認為他們的核心工作是為了支持還沒有正式承認存在的產品。將僱主的計劃公布在公共郵件列表上可能不是一個可行的選擇。在這種情況下，有必要考慮保密是否真的是必要的；通常不需要把開發計劃關在門內。

的確，有些情況下一家公司在開發過程的早期無法合法地披露其計劃。擁有經驗豐富的內核開發人員的公司可能選擇以開環的方式進行開發，前提是他們以後能夠避免嚴重的集成問題。對於沒有這種內部專業知識的公司，最好的選擇往往是聘請外部開發者根據保密協議審查計劃。Linux 基金會運行了一個 NDA 程序，旨在幫助解決這種情況；更多信息參見：

<http://www.linuxfoundation.org/nda/>

這種審查通常足以避免以後出現嚴重問題，而無需公開披露項目。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

---

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

---

**Original** Documentation/process/4.Coding.rst

**Translator** 時奎亮 Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

校譯 吳 想 成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)> 胡 皓 文 Hu Haowen  
<[src.res@email.cn](mailto:<src.res@email.cn>)>

## 使代碼正確

雖然一個堅實的、面向社區的設計過程有很多值得說道的，但是任何內核開發項目工作的證明都反映在代碼中。它是將由其他開發人員檢查併合並（或不併）到主線樹中的代碼。所以這段代碼的質量決定了項目的最終成功。

本節將檢查編碼過程。我們將從內核開發人員常犯的幾種錯誤開始。然後重點將轉移到正確的做法和相關有用的工具上。

## 陷阱

### 代碼風格

內核長期以來都有其標準的代碼風格，如<Documentation/process/coding-style.rst> 中所述。在多數時候，該文檔中描述的準則至多被認為是建議性的。因此，內核中存在大量不符合代碼風格準則的代碼。這種代碼的存在會給內核開發人員帶來兩方面的危害。

首先，相信內核代碼標準並不重要，也不強制執行。但事實上，如果沒有按照標準編寫代碼，那麼新代碼將很難加入到內核中；許多開發人員甚至會在審查代碼之前要求對代碼進行重新格式化。一個像內核這麼大的代碼庫需要一些統一格式的代碼，以使開發人員能夠快速理解其中的任何部分。所以再也經不起奇怪格式的代碼的折騰了。

內核的代碼風格偶爾會與僱主的強制風格發生衝突。在這種情況下，必須在代碼合併之前遵從內核代碼風格。將代碼放入內核意味著以多種方式放棄一定程度的控制權——包括控制代碼樣式。

另一個危害是認為已經在內核中的代碼迫切需要修復代碼樣式。開發者可能會開始編寫重新格式化補丁，作為熟悉開發過程的一種方式，或者作為將其名字寫入內核變更日誌的一種方式，或者兩者兼而有之。但是純代碼風格的修復被開發社區視為噪音，它們往往受到冷遇。因此，最好避免編寫這種類型的補丁。在由於其他原因處理一段代碼的同時順帶修復其樣式是很自然的，但是不應該僅為了更改代碼樣式而更改之。

代碼風格文檔也不應該被視為絕對不可違反的規則。如果有一個足夠的理由反對這種樣式（例如為了 80 列限制拆分行會導致可讀性大大降低），那麼就這樣做吧。

注意您還可以使用 clang-format 工具來幫助您處理這些規則，快速自動重新格式化部分代碼，和審閱完整的文件以發現代碼樣式錯誤、拼寫錯誤和可能的改進。它還可以方便地排序 #includes 、對齊變量/宏、重排文

本和其他類似任務。有關詳細信息，請參閱文檔 Documentation/process/clang-format.rst

### 抽象層

計算機科學教授教學生以靈活性和信息隱藏的名義廣泛使用抽象層。當然，內核廣泛地使用了抽象；任何涉及數百萬行代碼的項目都必須做到這一點以存續下來。但經驗表明，過度或過早的抽象可能和過早的優化一樣有害。抽象應用在適當層級，不要過度。

簡單點，先考慮一個調用時始終只有一個參數且總為零的函數。我們可以保留這個參數，以在需要使用它時提供的額外靈活性。不過，在那時實現了這個額外參數的代碼很有可能以某種從未被注意到的微妙方式被破壞——因為它從未被使用過。或者當需要額外的靈活性時，它並未以符合程式設計師當初期望的方式來實現。內核開發人員通常會提交補丁來刪除未使用的參數；一般來說，一開始就不應該添加這些參數。

隱藏硬體訪問的抽象層——通常為了允許大量的驅動程序兼容多個作業系統——尤其不受歡迎。這樣的層使代碼變得模糊，可能會造成性能損失；它們不屬於 Linux 內核。

另一方面，如果您發現自己從另一個內核子系統複製了大量的代碼，那麼是時候了解一下：是否需要將這些代碼中的部分提取到單獨的庫中，或者在更高的層次上實現這些功能。在整個內核中複製相同的代碼沒有價值。

### #ifdef 和預處理

C 預處理器似乎給一些 C 程式設計師帶來了強大的誘惑，他們認為它是一種將大量靈活性加入原始碼中的方法。但是預處理器不是 C，大量使用它會導致代碼對其他人來說更難閱讀，對編譯器來說更難檢查正確性。使用了大量預處理器幾乎總是代碼需要一些清理工作的標誌。

使用 #ifdef 的條件編譯實際上是一個強大的功能，它在內核中使用。但是很少有人希望看到代碼被鋪滿 #ifdef 塊。一般規定，ifdef 的使用應儘可能限制在頭文件中。條件編譯代碼可以限制函數，如果代碼不存在，這些函數就直接變成空的。然後編譯器將悄悄地優化對空函數的調用。使得代碼更加清晰，更容易理解。

C 預處理器宏存在許多危險性，包括可能對具有副作用且沒有類型安全的表達式進行多重評估。如果您試圖定義宏，請考慮創建一個內聯函數替代。結果相同的代碼，內聯函數更容易閱讀，不會多次計算其參數，並且允許編譯器對參數和返回值執行類型檢查。

### 內聯函數

不過，內聯函數本身也存在風險。程式設計師可以傾心於避免函數調用和用內聯函數填充源文件所固有的效率。然而，這些功能實際上會降低性能。因為它們的代碼在每個調用站點都被複製一遍，所以最終會增加編譯內核的大小。此外，這也對處理器的內存緩存造成壓力，從而大大降低執行速度。通常內聯函數應該非常小，而且相對較少。畢竟函數調用的成本並不低；大量創建內聯函數是過早優化的典型例子。

一般來說，內核程式設計師會自冒風險忽略緩存效果。在數據結構課程開頭中的經典時間/空間權衡通常不適用於當代硬體。空間就是時間，因為一個大的程序比一個更緊湊的程序運行得慢。

較新的編譯器越來越激進地決定一個給定函數是否應該內聯。因此，隨意放置使用「`inline`」關鍵字可能不僅僅是過度的，也可能是無用的。

## 鎖

2006 年 5 月，「`deviceescape`」網絡堆棧在前呼後擁下以 GPL 發布，並被納入主線內核。這是一個受歡迎的消息；Linux 中對無線網絡的支持充其量被認為是不合格的，而 `Deviceescape` 堆棧承諾修復這種情況。然而直到 2007 年 6 月（2.6.22），這段代碼才真正進入主線。發生了什麼？

這段代碼出現了許多閉門造車的跡象。但一個大麻煩是，它並不是為多處理器系統而設計。在合併這個網絡堆棧（現在稱為 `mac80211`）之前，需要對其進行一個鎖方案的改造。

曾經，Linux 內核代碼可以在不考慮多處理器系統所帶來的並發性問題的情況下進行開發。然而現在，這個文檔就是在雙核筆記本電腦上寫的。即使在單處理器系統上，為提高響應能力所做的工作也會提高內核內的並發性水平。編寫內核代碼而不考慮鎖的日子早已遠去。

可以由多個線程並發訪問的任何資源（數據結構、硬體寄存器等）必須由鎖保護。新的代碼應該謹記這一要求；事後修改鎖是一項相當困難的任務。內核開發人員應該花時間充分了解可用的鎖原語，以便為工作選擇正確的工具。對並發性缺乏關注的代碼很難進入主線。

## 回歸

最後一個值得一提的危險是回歸：它可能會引起導致現有用戶的某些東西中斷的改變（這也可能會帶來很大的改進）。這種變化被稱為「回歸」，回歸已經成為主線內核最不受歡迎的問題。除了少數例外情況，如果回歸不能及時修正，會導致回歸的修改將被取消。最好首先避免回歸發生。

人們常常爭論，如果回歸帶來的功能遠超過產生的問題，那麼回歸是否為可接受的。如果它破壞了一個系統卻為十個系統帶來新的功能，為何不改改態度呢？2007 年 7 月，Linus 對這個問題給出了最佳答案：

所以我們不會通過引入新問題來修復錯誤。這種方式是靠不住的，沒人知道是否真的有進展。是前進兩步、後退一步，還是前進一步、後退兩步？

[\(http://lwn.net/articles/243460/\)](http://lwn.net/articles/243460/)

特別不受歡迎的一種回歸類型是用戶空間 ABI 的任何變化。一旦接口被導出到用戶空間，就必須無限期地支持它。這一事實使得用戶空間接口的創建特別具有挑戰性：因為它們不能以不兼容的方式進行更改，所以必須一次就對。因此，用戶空間接口總是需要大量的思考、清晰的文檔和廣泛的審查。

## 代碼檢查工具

至少目前，編寫無錯誤代碼仍然是我們中很少人能達到的理想狀態。不過，我們希望做的是，在代碼進入主線內核之前，儘可能多地捕獲並修復這些錯誤。為此，內核開發人員已經提供了一系列令人印象深刻的工具，可以自動捕獲各種各樣的隱藏問題。計算機發現的任何問題都是一個以後不會困擾用戶的問題，因此，只要有可能，就應該使用自動化工具。

第一步是注意編譯器產生的警告。當前版本的 GCC 可以檢測（並警告）大量潛在錯誤。通常，這些警告都指向真正的問題。提交以供審閱的代碼一般不會產生任何編譯器警告。在消除警告時，注意了解真正的原因，並儘量避免僅「修復」使警告消失而不解決其原因。

請注意，並非所有編譯器警告都默認啓用。使用「`make KCFLAGS=-W`」構建內核以獲得完整集合。

內核提供了幾個配置選項，可以打開調試功能；大多數配置選項位於「kernel hacking」子菜單中。對於任何用於開發或測試目的的內核，都應該啓用其中幾個選項。特別是，您應該打開：

- `FRAME_WARN` 獲取大於給定數量的堆棧幀的警告。這些警告生成的輸出可能比較冗長，但您不必擔心來自內核其他部分的警告。
- `DEBUG_OBJECTS` 將添加代碼以跟蹤內核創建的各種對象的生命周期，並在出現問題時發出警告。如果你要添加創建（和導出）關於其自己的複雜對象的子系統，請考慮打開對象調試基礎結構的支持。
- `DEBUG_SLAB` 可以發現各種內存分配和使用錯誤；它應該用於大多數開發內核。
- `DEBUG_SPINLOCK`, `DEBUG_ATOMIC_SLEEP` 和 `DEBUG_MUTEXES` 會發現許多常見的鎖錯誤。

還有很多其他調試選項，其中一些將在下面討論。其中一些有顯著的性能影響，不應一直使用。在學習可用選項上花費一些時間，可能會在短期內得到許多回報。

其中一個較重的調試工具是鎖檢查器或「lockdep」。該工具將跟蹤系統中每個鎖（spinlock 或 mutex）的獲取和釋放、獲取鎖的相對順序、當前中斷環境等等。然後，它可以確保總是以相同的順序獲取鎖，相同的中斷假設適用於所有情況等等。換句話說，lockdep 可以找到許多導致系統死鎖的場景。在部署的系統中，這種問題可能會很痛苦（對於開發人員和用戶而言）；LockDep 允許提前以自動方式發現問題。具有任何類型的非普通鎖的代碼在提交合併前應在啓用 lockdep 的情況下運行測試。

作為一個勤奮的內核程式設計師，毫無疑問，您將檢查任何可能失敗的操作（如內存分配）的返回狀態。然而，事實上，最終的故障復現路徑可能完全沒有經過測試。未測試的代碼往往會出問題；如果所有這些錯誤處理路徑都被執行了幾次，那麼您可能對代碼更有信心。

內核提供了一個可以做到這一點的錯誤注入框架，特別是在涉及內存分配的情況下。啓用故障注入後，內存分配的可配置失敗的百分比；這些失敗可以限定在特定的代碼範圍內。在啓用了故障注入的情況下運行，程式設計師可以看到當情況惡化時代碼如何響應。有關如何使用此工具的詳細信息，請參閱 `Documentation/fault-injection/fault-injection.rst`。

「sparse」靜態分析工具可以發現其他類型的錯誤。`sparse` 可以警告程式設計師用戶空間和內核空間地址之間的混淆、大端序與小端序的混淆、在需要一組位標誌的地方傳遞整數值等等。`sparse` 必須單獨安裝（如果您的分發伺服器沒有將其打包，可以在 [https://sparse.wiki.kernel.org/index.php/Main\\_page](https://sparse.wiki.kernel.org/index.php/Main_page) 找到），然後可以通過在 `make` 命令中添加「`C=1`」在代碼上運行它。

「Coccinelle」工具 <http://coccinelle.lip6.fr/> 能夠發現各種潛在的編碼問題；它還可以為這些問題提出修復方案。在 scripts/coccinelle 目錄下已經打包了相當多的內核「語義補丁」；運行「make coccicheck」將運行這些語義補丁並報告發現的任何問題。有關詳細信息，請參閱 Documentation/dev-tools/coccinelle.rst 其他類型的可移植性錯誤最好通過為其他體系結構編譯代碼來發現。如果沒有 S/390 系統或 Blackfin 開發板，您仍然可以執行編譯步驟。可以在以下位置找到一大堆用於 x86 系統的交叉編譯器：

<https://www.kernel.org/pub/tools/crosstool/>

花一些時間安裝和使用這些編譯器將有助於避免以後的尷尬。

## 文檔

文檔通常比內核開發規則更為例外。即便如此，足夠的文檔將有助於簡化將新代碼合併到內核中的過程，使其他開發人員的生活更輕鬆，並對您的用戶有所幫助。在許多情況下，添加文檔已基本上是強制性的。

任何補丁的第一個文檔是其關聯的變更日誌。日誌條目應該描述正在解決的問題、解決方案的形式、處理補丁的人員、對性能的任何相關影響，以及理解補丁可能需要的任何其他內容。確保變更日誌說明了 \* 為什麼 \* 補丁值得應用；大量開發者未能提供這些信息。

任何添加新用戶空間接口的代碼——包括新的 sysfs 或/proc 文件——都應該包含該接口的文檔，該文檔使用戶空間開發人員能夠知道他們在使用什麼。請參閱 Documentation/ABI/README，了解如何此文檔格式以及需要提供哪些信息。

文檔 Documentation/admin-guide/kernel-parameters.rst 描述了內核的所有引導時間參數。任何添加新參數的補丁都應該向該文檔添加適當的條目。

任何新的配置選項都必須附有幫助文本，幫助文本需清楚地解釋這些選項以及用戶可能希望何時使用它們。

許多子系統的內部 API 信息通過專門格式化的注釋進行記錄；這些注釋可以通過「kernel-doc」腳本以多種方式提取和格式化。如果您在具有 kerneldoc 注釋的子系統中工作，則應該維護它們，並根據需要為外部可用的功能添加它們。即使在沒有如此記錄的領域中，為將來添加 kerneldoc 注釋也沒有壞處；實際上，這對於剛開始開發內核的人來說是一個有用的活動。這些注釋的格式以及如何創建 kerneldoc 模板的一些信息可以在 Documentation/doc-guide/ 上找到。

任何閱讀大量現有內核代碼的人都會注意到，注釋的缺失往往是最值得注意的。同時，對新代碼的要求比過去更高；合併未注釋的代碼將更加困難。這就是說，人們並不期望詳細注釋的代碼。代碼本身應該是自解釋的，注釋闡釋了更微妙的方面。

某些事情應該總是被注釋。使用內存屏障時，應附上一行文字，解釋為什麼需要設置內存屏障。數據結構的鎖規則通常需要在某個地方解釋。一般來說，主要數據結構需要全面的文檔。應該指出代碼中分立的位之間不明顯的依賴性。任何可能誘使代碼管理人進行錯誤的「清理」的事情都需要一個注釋來說明為什麼要這樣做。等等。

### 內部 API 更改

內核提供給用戶空間的二進位接口不能被破壞，除非逼不得已。而內核的內部編程接口是高度流動的，當需要時可以更改。如果你發現自己不得不處理一個內核 API，或者僅僅因為它不滿足你的需求導致無法使用特定的功能，這可能是 API 需要改變的一個標誌。作為內核開發人員，您有權進行此類更改。

的確可以進行 API 更改，但更改必須是合理的。因此任何進行內部 API 更改的補丁都應該附帶關於更改內容和必要原因的描述。這種變化也應該拆分成一個單獨的補丁，而不是埋在一個更大的補丁中。

另一個要點是，更改內部 API 的開發人員通常要負責修復內核樹中被更改破壞的任何代碼。對於一個廣泛使用的函數，這個責任可以導致成百上千的變化，其中許多變化可能與其他開發人員正在做的工作相衝突。不用說，這可能是一項大工程，所以最好確保理由是可靠的。請注意，coccinelle 工具可以幫助進行廣泛的 API 更改。

在進行不兼容的 API 更改時，應儘可能確保編譯器捕獲未更新的代碼。這將幫助您確保找到該接口的樹內用處。它還將警告開發人員樹外代碼存在他們需要響應的更改。支持樹外代碼不是內核開發人員需要擔心的事情，但是我們也不必使樹外開發人員的生活有不必要的困難。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

---

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

---

**Original** Documentation/process/5.Posting.rst

**Translator** 時奎亮 Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

校譯 吳 想 成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)> 胡 皓 文 Hu Haowen  
<[src.res@email.cn](mailto:src.res@email.cn)>

### 發布補丁

您的工作遲早會準備好提交給社區進行審查，並最終包含到主線內核中。毫不稀奇，內核開發社區已經發展出一套用於發布補丁的約定和過程；遵循這些約定和過程將使參與其中的每個人的生活更加輕鬆。本文檔試圖描述這些約定的部分細節；更多信息也可在以下文檔中找到[Documentation/translations/zh\\_TW/process/submitting-patches.rst](#)，[Documentation/translations/zh\\_TW/process/submitting-drivers.rst](#) 和[Documentation/translations/zh\\_TW/process/submit-checklist.rst](#)。

## 何時郵寄

在補丁完全「準備好」之前，避免發布補丁是一種持續的誘惑。對於簡單的補丁，這不是問題。但是如果正在完成的工作很複雜，那麼在工作完成之前從社區獲得反饋就可以獲得很多好處。因此，您應該考慮發布正在進行的工作，甚至維護一個可用的 Git 樹，以便感興趣的開發人員可以隨時趕上您的工作。

當發布中有尚未準備好被包含的代碼，最好在發布中說明。還應提及任何有待完成的主要工作和任何已知問題。很少有人會願意看那些被認為是半生不熟的補丁，但是那些願意的人會帶著他們的點子來一起幫助你把工作推向正確的方向。

## 創建補丁之前

在考慮將補丁發送到開發社區之前，有許多事情應該做。包括：

- 儘可能地測試代碼。利用內核的調試工具，確保內核使用了所有可能的配置選項組合進行構建，使用交叉編譯器為不同的體系結構進行構建等。
- 確保您的代碼符合內核代碼風格指南。
- 您的更改是否具有性能影響？如果是這樣，您應該運行基準測試來顯示您的變更的影響（或好處）；結果的摘要應該包含在補丁中。
- 確保您有權發布代碼。如果這項工作是為僱主完成的，僱主對這項工作具有所有權，並且必須同意根據 GPL 對其進行發布。

一般來說，在發布代碼之前進行一些額外的思考，幾乎總是能在短時間內得到回報。

## 補丁準備

準備補丁發布的工作量可能很驚人，但在此嘗試節省時間通常是不明智的，即使在短期內亦然。

必須針對內核的特定版本準備補丁。一般來說，補丁應該基於 Linus 的 Git 樹中的當前主線。當以主線為基礎時，請從一個衆所周知的發布點開始——如穩定版本或 -rc 版本發布點——而不是在一個任意的主線分支點。

也可能需要針對-mm、linux-next 或子系統樹生成版本，以便於更廣泛的測試和審查。根據補丁的區域以及其他地方的情況，針對其他樹建立的補丁可能需要大量的工作來解決衝突和處理 API 更改。

只有最簡單的更改才應格式化為單個補丁；其他所有更改都應作為一系列邏輯更改進行。分割補丁是一門藝術；一些開發人員花了很長時間來弄清楚如何按照社區期望的方式來分割。不過，這些經驗法則也許有幫助：

- 您發布的補丁系列幾乎肯定不會是開發過程中版本控制系統中的一系列更改。相反，需要對您所做更改的最終形式加以考慮，然後以有意義的方式進行拆分。開發人員對離散的、自包含的更改感興趣，而不是您創造這些更改的原始路徑。
- 每個邏輯上獨立的變更都應該格式化為單獨的補丁。這些更改可以是小的（如「向此結構體添加欄位」）或大的（如添加一個重要的新驅動程序），但它們在概念上應該是小的，並且可以在一行內簡述。每個補丁都應該做一個特定的、可以單獨檢查並驗證它所做的事情的更改。

- 換種方式重申上述準則，也就是說：不要在同一補丁中混合不同類型的更改。如果一個補丁修復了一個關鍵的安全漏洞，又重新排列了一些結構，還重新格式化了代碼，那麼它很有可能會被忽略，從而導致重要的修復丟失。
- 每個補丁都應該能創建一個可以正確地構建和運行的內核；如果補丁系列在中間被斷開，那麼結果仍應是一個正常工作的內核。部分應用一系列補丁是使用「git bisect」工具查找回歸的一個常見場景；如果結果是一個損壞的內核，那麼將使那些從事追蹤問題的高尚工作的開發人員和用戶的生活更加艱難。
- 不要過分分割。一位開發人員曾經將一組針對單個文件的編輯分成 500 個單獨的補丁發布，這並沒有使他成為內核郵件列表中最受歡迎的人。一個補丁可以相當大，只要它仍然包含一個單一的 邏輯變更。
- 用一系列補丁添加一個全新的基礎設施，但是該設施在系列中的最後一個補丁啓用整個變更之前不能使用，這看起來很誘人。如果可能的話，應該避免這種誘惑；如果這個系列增加了回歸，那麼二分法將指出最後一個補丁是導致問題的補丁，即使真正的 bug 在其他地方。只要有可能，添加新代碼的補丁程序應該立即激活該代碼。

創建完美補丁系列的工作可能是一個令人沮喪的過程，在完成「真正的工作」之後需要花費大量的時間和思考。但是如果做得好，花費的時間就是值得的。

### 補丁格式和更改日誌

所以現在你有了一系列完美的補丁可以發布，但是這項工作還沒有完成。每個補丁都需要被格式化成一條消息，以快速而清晰地將其目的傳達到世界其他地方。為此，每個補丁將由以下部分組成：

- 可選的「From」行，表明補丁作者。只有當你通過電子郵件發送別人的補丁時，這一行才是必須的，但是為防止疑問加上它也不會有什麼壞處。
- 一行描述，說明補丁的作用。對於在沒有其他上下文的情況下看到該消息的讀者來說，該消息應足以確定修補程序的範圍；此行將顯示在「short form（簡短格式）」變更日誌中。此消息通常需要先加上子系統名稱前綴，然後是補丁的目的。例如：

```
gpio: fix build on CONFIG_GPIO_SYSFS=n
```

- 一行空白，後接補丁內容的詳細描述。此描述可以是任意需要的長度；它應該說明補丁的作用以及為什麼它應該應用於內核。
- 一個或多個標記行，至少有一個由補丁作者的 Signed-off-by 簽名。標記將在下面詳細描述。

上面的項目一起構成補丁的變更日誌。寫一則好的變更日誌是一門至關重要但常常被忽視的藝術；值得花一點時間來討論這個問題。當你編寫變更日誌時，你應該記住有很多不同的人會讀你的話。其中包括子系統維護人員和審查人員，他們需要決定是否應該合併補丁，分銷商和其他維護人員試圖決定是否應該將補丁反向移植到其他內核，缺陷搜尋人員想知道補丁是否導致他們正在追查的問題，以及想知道內核如何變化的用戶等等。一個好的變更日誌以最直接和最簡潔的方式向所有這些人傳達所需的信息。

在結尾，總結行應該描述變更的影響和動機，以及在一行約束條件下可能發生的變化。然後，詳細的描述可以詳述這些主題，並提供任何需要的附加信息。如果補丁修復了一個缺陷，請引用引入該缺陷的提交（如果可能，請在引用提交時同時提供其 id 和標題）。如果某個問題與特定的日誌或編譯器輸出相關聯，請包含該輸出以幫助其他人搜索同一問題的解決方案。如果更改是為了支持以後補丁中的其他更改，那麼應當說明。如

果更改了內部 API，請詳細說明這些更改以及其他開發人員應該如何響應。一般來說，你越把自己放在每個閱讀你變更日誌的人的位置上，變更日誌（和內核作為一個整體）就越好。

不消說，變更日誌是將變更提交到版本控制系統時使用的文本。接下來將是：

- 補丁本身，採用統一的（「-u」）補丁格式。使用「-p」選項來 diff 將使函數名與更改相關聯，從而使結果補丁更容易被其他人讀取。

您應該避免在補丁中包括與更改不相關文件（例如，構建過程生成的文件或編輯器備份文件）。文檔目錄中的「`dontdiff`」文件在這方面有幫助；使用「-X」選項將其傳遞給 diff。

上面提到的標籤（tag）用於描述各種開發人員如何與這個補丁的開發相關聯。[Documentation/translations/zh\\_TW/process/submitting-patches.rst](#) 文檔中對它們進行了詳細描述；下面是一個簡短的總結。每一行的格式如下：

```
tag: Full Name <email address> optional-other-stuff
```

常用的標籤有：

- **Signed-off-by:** 這是一個開發人員的證明，證明他或她有權提交補丁以包含到內核中。這表明同意開發者來源認證協議，其全文見[Documentation/translations/zh\\_TW/process/submitting-patches.rst](#) 如果沒有合適的簽字，則不能合併到主線中。
- **Co-developed-by:** 聲明補丁是由多個開發人員共同創建的；當幾個人在一個補丁上工作時，它用於給出共同作者（除了 From: 所給出的作者之外）。由於 Co-developed-by: 表示作者身份，所以每個共同開發人，必須緊跟在相關合作作者的 Signed-off-by 之後。具體內容和示例見以下文件[Documentation/translations/zh\\_TW/process/submitting-patches.rst](#)
- **Acked-by:** 表示另一個開發人員（通常是相關代碼的維護人員）同意補丁適合包含在內核中。
- **Tested-by:** 聲明某人已經測試了補丁並確認它可以工作。
- **Reviewed-by:** 表示某開發人員已經審查了補丁的正確性；有關詳細信息，請參閱[Documentation/translations/zh\\_TW/process/submitting-patches.rst](#)
- **Reported-by:** 指定報告此補丁修復的問題的用戶；此標記用於表示感謝。
- **Cc:** 指定某人收到了補丁的副本，並有機會對此發表評論。

在補丁中添加標籤時要小心：只有 Cc: 才適合在沒有指定人員明確許可的情況下添加。

## 發送補丁

在寄出補丁之前，您還需要注意以下幾點：

- 您確定您的郵件發送程序不會損壞補丁嗎？被郵件客戶端更改空白或修飾了行的補丁無法被另一端接受，並且通常不會進行任何詳細檢查。如果有任何疑問，先把補丁寄給你自己，讓你自己確定它是完好無損的。

[Documentation/translations/zh\\_TW/process/email-clients.rst](#) 提供了一些有用的提示，可以讓特定的郵件客戶端正常發送補丁。

- 你確定你的補丁沒有荒唐的錯誤嗎？您應該始終通過 scripts/checkpatch.pl 檢查補丁程序，並解決它提出的問題。請記住，checkpatch.pl，雖然體現了對內核補丁應該是什麼樣的大量思考，但它並不比您聰明。如果修復 checkpatch.pl 紿的問題會使代碼變得更糟，請不要這樣做。

補丁應始終以純文本形式發送。請不要將它們作為附件發送；這使得審閱者在答覆中更難引用補丁的部分。相反，只需將補丁直接放到您的消息中。

寄出補丁時，重要的是將副本發送給任何可能感興趣的人。與其他一些項目不同，內核鼓勵人們甚至錯誤地發送過多的副本；不要假定相關人員會看到您在郵件列表中的發布。尤其是，副本應發送至：

- 受影響子系統的維護人員。如前所述，維護人員文件是查找這些人員的首選地方。
- 其他在同一領域工作的開發人員，尤其是那些現在可能在那裡工作的開發人員。使用 git 查看還有誰修改了您正在處理的文件，這很有幫助。
- 如果您對某錯誤報告或功能請求做出響應，也可以抄送原始發送人。
- 將副本發送到相關郵件列表，或者若無相關列表，則發送到 linux-kernel 列表。
- 如果您正在修復一個缺陷，請考慮該修復是否應進入下一個穩定更新。如果是這樣，補丁副本也應發到 stable@vger.kernel.org。另外，在補丁本身的標籤中添加一個「Cc: stable@vger.kernel.org」；這將使穩定版團隊在修復進入主線時收到通知。

當為一個補丁選擇接收者時，最好清楚你認為誰最終會接受這個補丁並將其合併。雖然可以將補丁直接發給 Linus Torvalds 並讓他合併，但通常情況下不會這樣做。Linus 很忙，並且有子系統維護人員負責監視內核的特定部分。通常您會希望維護人員合併您的補丁。如果沒有明顯的維護人員，Andrew Morton 通常是最後的補丁接收者。

補丁需要好的主題行。補丁主題行的規範格式如下：

```
[PATCH nn/mm] subsys: one-line description of the patch
```

其中「nn」是補丁的序號，「mm」是系列中補丁的總數，「subsys」是受影響子系統的名稱。當然，一個單獨的補丁可以省略 nn/mm。

如果您有一系列重要的補丁，那麼通常發送一個簡介作為第〇部分。不過，這個約定並沒有得到普遍遵循；如果您使用它，請記住簡介中的信息不會進入內核變更日誌。因此，請確保補丁本身具有完整的變更日誌信息。

一般來說，多部分補丁的第二部分和後續部分應作為對第一部分的回覆發送，以便它們在接收端都連接在一起。像 git 和 coilt 這樣的工具有命令，可以通過適當的線程發送一組補丁。但是，如果您有一長串補丁，並正使用 git，請不要使用--chain-reply-to 選項，以避免創建過深的嵌套。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

### Original Documentation/process/6.Followthrough.rst

**Translator** 時奎亮 Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

校譯 吳 想 成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)> 胡 皓 文 Hu Haowen  
<[src.res@email.cn](mailto:<src.res@email.cn>)>

## 跟進

此時，您已經遵循了到目前為止給出的指導方針，並且，隨著您自己的工程技能的增加，已經發布了一系列完美的補丁。即使是經驗豐富的內核開發人員也能犯的最大錯誤之一是，認為他們的工作現在已經完成了。事實上，發布補丁意味著進入流程的下一個階段，可能還需要做很多工作。

一個補丁在首次發布時就非常出色、沒有改進的餘地，這是很罕見的。內核開發流程已認識到這一事實，因此它非常注重對已發布代碼的改進。作為代碼的作者，您應該與內核社區合作，以確保您的代碼符合內核的質量標準。如果不參與這個過程，很可能會無法將補丁合併到主線中。

## 與審閱者合作

任何意義上的補丁都會導致其他開發人員在審查代碼時發表大量評論。對於許多開發人員來說，與審閱人員合作可能是內核開發過程中最令人生畏的部分。但是如果你記住一些事情，生活會變得容易得多：

- 如果你已經很好地解釋了你的補丁，審閱人員會理解它的價值，以及為什麼你會費盡心思去寫它。但是這個並不能阻止他們提出一個基本的問題：在五年或十年後維護含有此代碼的內核會怎麼樣？你可能被要求做出的許多改變——從編碼風格的調整到大量的重寫——都來自於對 Linux 的理解，即從現在起十年後，Linux 仍將在開發中。
- 代碼審查是一項艱苦的工作，這是一項相對吃力不討好的工作；人們記得誰編寫了內核代碼，但對於那些審查它的人來說，幾乎沒有什麼長久的名聲。因此，審閱人員可能會變得暴躁，尤其是當他們看到同樣的錯誤被一遍又一遍地犯下時。如果你得到了一個看起來憤怒、侮辱或完全冒犯你的評論，請抑制以同樣方式回應的衝動。代碼審查是關於代碼的，而不是關於人的，代碼審閱人員不會親自攻擊您。
- 同樣，代碼審閱人員也不想以犧牲你僱主的利益為代價來宣傳他們僱主的議程。內核開發人員通常希望今後幾年能在內核上工作，但他們明白他們的僱主可能會改變。他們真的，幾乎毫無例外地，致力於創造他們所能做到的最好的內核；他們並沒有試圖給僱主的競爭對手造成不適。

所有這些歸根結底就是，當審閱者向您發送評論時，您需要注意他們正在進行的技術評論。不要讓他們的表達方式或你自己的驕傲阻止此事。當你在一個補丁上得到評論時，花點時間去理解評論人想說什麼。如果可能的話，請修復審閱者要求您修復的內容。然後回覆審閱者：謝謝他們，並描述你將如何回答他們的問題。

請注意，您不必同意審閱者建議的每個更改。如果您認為審閱者誤解了您的代碼，請解釋到底發生了什麼。如果您對建議的更改有技術上的異議，請描述它並證明您對該問題的解決方案是正確的。如果你的解釋有道

理，審閱者會接受的。不過，如果你的解釋證明缺乏說服力，尤其是當其他人開始同意審稿人的觀點時，請花些時間重新考慮一下。你很容易對自己解決問題的方法視而不見，以至於你沒有意識到某些東西完全是錯誤的，或者你甚至沒有解決正確的問題。

Andrew Morton 建議，每一個不會導致代碼更改的審閱評論都應該產生一個額外的代碼注釋；這可以幫助未來的審閱人員避免第一次出現的問題。

一個致命的錯誤是忽視評論，希望它們會消失。它們不會走的。如果您在沒有對之前收到的評論做出響應的情況下重新發布代碼，那麼很可能會發現補丁毫無用處。

說到重新發布代碼：請記住，審閱者不會記住您上次發布的代碼的所有細節。因此，提醒審閱人員以前提出的問題以及您如何處理這些問題總是一個好主意；補丁變更日誌是提供此類信息的好地方。審閱者不必搜索列表檔案來熟悉上次所說的內容；如果您幫助他們直接開始，當他們重新查看您的代碼時，心情會更好。

如果你已經試著做正確的事情，但事情仍然沒有進展呢？大多數技術上的分歧都可以通過討論來解決，但有時人們仍需要做出決定。如果你真的認為這個決定對你不利，你可以試著向有更高權力的人上訴。對於本文，更高權力的人是 Andrew Morton。Andrew 在內核開發社區中非常受尊敬；他經常為似乎被絕望阻塞的事情清障。儘管如此，不應輕易就直接找 Andrew，也不應在所有其他替代方案都被嘗試之前找他。當然，記住，他也可能不同意你的意見。

### 接下來會發生什麼

如果一個補丁被認為適合添加到內核中，並且大多數審查問題得到解決，下一步通常是進入子系統維護人員的樹中。工作方式因子系統而異；每個維護人員都有自己的工作方式。特別是可能有不止一棵樹——也許一棵樹專門用於計劃下一個合併窗口的補丁，另一棵樹用於長期工作。

對於應用到不屬於明顯子系統樹（例如內存管理修補程序）的區域的修補程序，默認樹通常上溯到-mm。影響多個子系統的補丁也可以最終進入-mm 樹。

包含在子系統樹中可以提高補丁的可見性。現在，使用該樹的其他開發人員將默認獲得補丁。子系統樹通常也為 Linux 提供支持，使其內容對整個開發社區可見。在這一點上，您很可能會從一組新的審閱者那裡得到更多的評論；這些評論需要像上一輪那樣得到回應。

在這時也會發生點什麼，這取決於你的補丁的性質，是否與其他人正在做的工作發生衝突。在最壞的情況下，嚴重的補丁衝突可能會導致一些工作被擱置，以便剩餘的補丁可以成形併合並。另一些時候，衝突解決將涉及到與其他開發人員合作，可能還會在樹之間移動一些補丁，以確保所有的應用都是乾淨的。這項工作可能是一件痛苦的事情，但也需慶幸現在的幸福：在 linux-next 樹出現之前，這些衝突通常只在合併窗口中出現，必須迅速解決。現在可以在合併窗口打開之前的空閒時間解決這些問題。

有朝一日，如果一切順利，您將登錄並看到您的補丁已經合併到主線內核中。祝賀你！然而，一旦慶祝完了（並且您已經將自己添加到維護人員文件中），就一定要記住一個重要的小事實：工作仍然沒有完成。併入主線也帶來了它的挑戰。

首先，補丁的可見性再次提高。可能會有以前不知道這個補丁的開發者的新一輪評論。忽略它們可能很有誘惑力，因為您的代碼不再存在任何被合併的問題。但是，要抵制這種誘惑，您仍然需要對有問題或建議的開發人員作出響應。

不過，更重要的是：將代碼包含在主線中會將代碼交給更多的一些測試人員。即使您為尚未可用的硬體提供了驅動程序，您也會驚訝於有多少人會將您的代碼構建到內核中。當然，如果有測試人員，也可能會有錯誤報告。

最糟糕的錯誤報告是回歸。如果你的補丁導致回歸，你會發現多到讓你不舒服的眼睛盯著你；回歸需要儘快修復。如果您不願意或無法修復回歸（其他人都不會為您修復），那麼在穩定期內，您的補丁幾乎肯定會被移除。除了否定您為使補丁進入主線所做的所有工作之外，如果由於未能修復回歸而取消補丁，很可能會使將來的工作更難被合併。

在處理完任何回歸之後，可能還有其他普通缺陷需要處理。穩定期是修復這些錯誤並確保代碼在主線內核版本中的首次發布儘可能可靠的最好機會。所以，請回應錯誤報告，並儘可能解決問題。這就是穩定期的目的；一旦解決了舊補丁的任何問題，就可以開始盡情創建新補丁。

別忘了，還有其他節點也可能會創建缺陷報告：下一個主線穩定版本，當著名的發行商選擇包含您補丁的內核版本時等等。繼續響應這些報告是您工作的基本素養。但是如果這不能提供足夠的動機，那麼也需要考慮：開發社區會記住那些在合併後對代碼失去興趣的開發人員。下一次你發布補丁時，他們會以你以後不會持續維護它為前提來評估它。

## 其他可能發生的事情

某天，當你打開你的郵件客戶端時，看到有人給你寄了一個代碼補丁。畢竟，這是讓您的代碼公開存在的好處之一。如果您同意這個補丁，您可以將它轉發給子系統維護人員（確保包含一個正確的 From: 行，這樣屬性是正確的，並添加一個您自己的 signoff），或者回復一個 Acked-by: 讓原始發送者向上發送它。

如果您不同意補丁，請禮貌地回復，解釋原因。如果可能的話，告訴作者需要做哪些更改才能讓您接受補丁。合併代碼的編寫者和維護者所反對的補丁的確存在著一定的阻力，但僅此而已。如果你被認為不必要的阻礙了好的工作，那麼這些補丁最終會繞過你並進入主線。在 Linux 內核中，沒有人對任何代碼擁有絕對的否決權。可能除了 Linus。

在非常罕見的情況下，您可能會看到完全不同的東西：另一個開發人員發布了針對您的問題的不同解決方案。在這時，兩個補丁之一可能不會被合併，「我的補丁首先發布」不被認為是一個令人信服的技術論據。如果有別人的補丁取代了你的補丁而進入了主線，那麼只有一種方法可以回應你：很高興你的問題解決了，請繼續工作吧。以這種方式把某人的工作推到一邊可能導致傷心和氣餒，但是社區會記住你的反應，即使很久以後他們已經忘記了誰的補丁真正被合併。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

**Original** Documentation/process/7.AdvancedTopics.rst

**Translator** 時奎亮 Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

校譯 吳 想 成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)> 胡 皓 文 Hu Haowen  
<[src.res@email.cn](mailto:src.res@email.cn)>

### 高級主題

現在，希望您能夠掌握開發流程的工作方式。然而，還有更多的東西要學！本節將介紹一些主題，這些主題對希望成為 Linux 內核開發過程常規部分的開發人員有幫助。

#### 使用 Git 管理補丁

內核使用分布式版本控制始於 2002 年初，當時 Linus 首次開始使用專有的 Bitkeeper 應用程序。雖然 BitKeeper 存在爭議，但它所體現的軟體版本管理方法卻肯定不是。分布式版本控制可以立即加速內核開發項目。現在有好幾種免費的 BitKeeper 替代品。但無論好壞，內核項目都已經選擇了 Git 作為其工具。

使用 Git 管理補丁可以使開發人員的生活更加輕鬆，尤其是隨著補丁數量的增長。Git 也有其粗糙的邊角和一定的危險性，它是一個年輕和強大的工具，仍然在其開發人員完善中。本文檔不會試圖教會讀者如何使用 git；這會是個巨長的文檔。相反，這裡的重點將是 Git 如何特別適合內核開發過程。想要加快用 Git 速度的開發人員可以在以下網站上找到更多信息：

<https://git-scm.com/>

<https://www.kernel.org/pub/software/scm/git/docs/user-manual.html>

同時網上也能找到各種各樣的教程。

在嘗試使用它生成補丁供他人使用之前，第一要務是閱讀上述網頁，對 Git 的工作方式有一個紮實的了解。使用 Git 的開發人員應能進行拉取主線存儲庫的副本，查詢修訂歷史，提交對樹的更改，使用分支等操作。了解 Git 用於重寫歷史的工具（如 rebase）也很有用。Git 有自己的術語和概念；Git 的新用戶應該了解引用、遠程分支、索引、快進合併、推拉、游離頭等。一開始可能有點嚇人，但這些概念不難通過一點學習來理解。

使用 git 生成通過電子郵件提交的補丁是提高速度的一個很好的練習。

當您準備好開始建立 Git 樹供其他人查看時，無疑需要一個可以從中拉取的伺服器。如果您有一個可以訪問網際網路的系統，那麼使用 git-daemon 設置這樣的伺服器相對簡單。同時，免費的公共託管網站（例如 github）也開始出現在網絡上。成熟的開發人員可以在 kernel.org 上獲得一個帳戶，但這些帳戶並不容易得到；更多有關信息，請參閱 <https://kernel.org/faq/>。

正常的 Git 工作流程涉及到許多分支的使用。每一條開發線都可以分為單獨的「主題分支」，並獨立維護。Git 的分支很容易使用，沒有理由不使用它們。而且，在任何情況下，您都不應該在任何您打算讓其他人從中拉取的分支中進行開發。應該小心地創建公開可用的分支；當開發分支處於完整狀態並已準備好時（而不是之前）才合併開發分支的補丁。

Git 提供了一些強大的工具，可以讓您重寫開發歷史。一個不方便的補丁（比如說，一個打破二分法的補丁，或者有其他一些明顯的缺陷）可以在適當的位置修復，或者完全從歷史中消失。一個補丁系列可以被重寫，就好像它是在今天的主線上寫的一樣，即使你已經花了幾個月的時間在寫它。可以透明地將更改從一個分支轉移到另一個分支。等等。明智地使用 git 修改歷史的能力可以幫助創建問題更少的乾淨補丁集。

然而，過度使用這種功能可能會導致其他問題，而不僅僅是對創建完美項目歷史的簡單癡迷。重寫歷史將重寫該歷史中包含的更改，將經過測試（希望如此）的內核樹變為未經測試的內核樹。除此之外，如果開發人員沒有共享項目歷史，他們就無法輕鬆地協作；如果您重寫了其他開發人員拉入他們存儲庫的歷史，您將使這些開發人員的生活更加困難。因此，這裡有一個簡單的經驗法則：被導出到其他地方的歷史在此後通常被認為是不可變的。

因此，一旦將一組更改推送到公開可用的伺服器上，就不應該重寫這些更改。如果您嘗試強制進行無法快進合併的更改（即不共享同一歷史記錄的更改），Git 將嘗試強制執行此規則。這可能覆蓋檢查，有時甚至需要重寫導出的樹。在樹之間移動變更集以避免 linux-next 中的衝突就是一個例子。但這種行為應該是罕見的。這就是為什麼開發應該在私有分支中進行（必要時可以重寫）並且只有在公共分支處於合理的較新狀態時才轉移到公共分支中的原因之一。

當主線（或其他一組變更所基於的樹）前進時，很容易與該樹合併以保持領先地位。對於一個私有的分支，rebasing 可能是一個很容易跟上另一棵樹的方法，但是一旦一棵樹被導出到外界，rebasing 就不可取了。一旦發生這種情況，就必須進行完全合併（merge）。合併有時是很有意義的，但是過於頻繁的合併會不必要的擾亂歷史。在這種情況下建議的做法是不要頻繁合併，通常只在特定的發布點（如主線-rc 發布）合併。如果您對特定的更改感到緊張，則可以始終在私有分支中執行測試合併。在這種情況下，git 「rerere」工具很有用；它能記住合併衝突是如何解決的，這樣您就不必重複相同的工作。

關於 Git 這樣的工具的一個最大的反覆抱怨是：補丁從一個存儲庫到另一個存儲庫的大量移動使得很容易陷入錯誤建議的變更中，這些變更避開審查雷達進入主線。當內核開發人員看到這種情況發生時，他們往往會感到不高興；在 Git 樹上放置未審閱或主題外的補丁可能會影響您將來讓樹被拉取的能力。引用 Linus 的話：

你可以給我發補丁，但當我從你那裡拉取一個 Git 補丁時，我需要知道你清楚自己在做什麼，我需要能夠相信事情而 \* 無需 \* 手動檢查每個單獨的更改。

(<http://lwn.net/articles/224135/>)。

爲了避免這種情況，請確保給定分支中的所有補丁都與相關主題緊密相關；「驅動程序修復」分支不應更改核心內存管理代碼。而且，最重要的是，不要使用 Git 樹來繞過審查過程。不時的將樹的摘要發布到相關的列表中，在合適時候請求 linux-next 中包含該樹。

如果其他人開始發送補丁以包含到您的樹中，不要忘記審閱它們。還要確保您維護正確的作者信息；git 「am」工具在這方面做得最好，但是如果補丁通過第三方轉發給您，您可能需要在補丁中添加「From:」行。

請求拉取時，請務必提供所有相關信息：樹的位置、要拉取的分支以及拉取將導致的更改。在這方面 git request-pull 命令非常有用；它將按照其他開發人員所期望的格式化請求，並檢查以確保您已記得將這些更改推送到公共伺服器。

### 審閱補丁

一些讀者顯然會反對將本節與「高級主題」放在一起，因為即使是剛開始的內核開發人員也應該審閱補丁。當然，沒有比查看其他人發布的代碼更好的方法來學習如何在內核環境中編程了。此外，審閱者永遠供不應求；通過審閱代碼，您可以對整個流程做出重大貢獻。

審查代碼可能是一副令人生畏的圖景，特別是對一個新的內核開發人員來說，他們可能會對公開詢問代碼感到緊張，而這些代碼是由那些有更多經驗的人發布的。不過，即使是最有經驗的開發人員編寫的代碼也可以得到改進。也許對（所有）審閱者最好的建議是：把審閱評論當成問題而不是批評。詢問「在這條路徑中如何釋放鎖？」總是比說「這裡的鎖是錯誤的」更好。

不同的開發人員將從不同的角度審查代碼。部分人會主要關注代碼風格以及代碼行是否有尾隨空格。其他人會主要關注補丁作為一個整體實現的變更是否對內核有好處。同時也有人會檢查是否存在鎖問題、堆棧使用過度、可能的安全問題、在其他地方發現的代碼重複、足夠的文檔、對性能的不利影響、用戶空間 ABI 更改等。所有類型的檢查，只要它們能引導更好的代碼進入內核，都是受歡迎和值得的。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

**Original** Documentation/process/8.Conclusion.rst

**Translator** 時奎亮 Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>

校譯 吳 想 成 Wu XiangCheng <[bobwxc@email.cn](mailto:bobwxc@email.cn)> 胡 皓 文 Hu Haowen  
<[src.res@email.cn](mailto:<src.res@email.cn>)>

### 更多信息

關於 Linux 內核開發和相關主題的信息來源很多。首先是在內核原始碼分發中找到的文檔目錄。頂級 [Documentation/translations/zh\\_TW/process/howto.rst](#) 文件是一個重要的起點；[Documentation/translations/zh\\_TW/process/submitting-patches.rst](#) 和 [Documentation/translations/zh\\_TW/process/submitting-drivers.rst](#) 也是所有內核開發人員都應該閱讀的內容。許多內部內核 API 都是使用 kerneldoc 機制記錄的；「make htmldocs」或「make pdfdocs」可用於以 HTML 或 PDF 格式生成這些文檔（儘管某些發行版提供的 tex 版本會遇到內部限制，無法正確處理文檔）。

不同的網站在各個細節層次上討論內核開發。本文作者想謙虛地建議用 <https://lwn.net/> 作為來源；有關許多特定內核主題的信息可以通過以下網址的 LWN 內核索引找到：

<http://lwn.net/kernel/index/>

除此之外，內核開發人員的一個寶貴資源是：

<https://kernelnewbies.org/>

當然，也不應該忘記 <https://kernel.org/>，這是內核發布信息的最終位置。

關於內核開發有很多書：

《Linux 設備驅動程序》第三版 (Jonathan Corbet、Alessandro Rubini 和 Greg Kroah Hartman) 線上版本在 <http://lwn.net/kernel/ldd3/>

《Linux 內核設計與實現》(Robert Love)

《深入理解 Linux 內核》(Daniel Bovet 和 Marco Cesati)

然而，所有這些書都有一個共同的缺點：它們上架時就往往有些過時，而且已經上架一段時間了。不過，在那裡還是可以找到相當多的好信息。

有關 git 的文檔，請訪問：

<https://www.kernel.org/pub/software/scm/git/docs/>

<https://www.kernel.org/pub/software/scm/git/docs/user-manual.html>

## 結論

祝賀所有通過這篇冗長的文檔的人。希望它能夠幫助您理解 Linux 內核是如何開發的，以及您如何參與這個過程。

最後，重要的是參與。任何開源軟體項目都不會超過其貢獻者投入其中的總和。Linux 內核的發展速度和以前一樣快，因為它得到了大量開發人員的幫助，他們都在努力使它變得更好。內核是一個最成功的例子，說明了當成千上萬的人為了一個共同的目標一起工作時，可以做出什麼。

不過，內核總是可以從更大的開發人員基礎中獲益。總有更多的工作要做。但是同樣重要的是，Linux 生態系統中的大多數其他參與者可以通過為內核做出貢獻而受益。使代碼進入主線是提高代碼質量、降低維護和分發成本、提高對內核開發方向的影響程度等的關鍵。這是一種共贏的局面。啓動你的編輯器，來加入我們吧；你會非常受歡迎的。

本文檔的目的是幫助開發人員（及其經理）以最小的挫折感與開發社區合作。它試圖記錄這個社區如何以一種不熟悉 Linux 內核開發（或者實際上是自由軟體開發）的人可以訪問的方式工作。雖然這裡有一些技術資料，但這是一個面向過程的討論，不需要深入了解內核編程就可以理解。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

---

### Original Documentation/process/email-clients.rst

譯者：

中文版維護者： 賈威威 Harry Wei <[harryxiyou@gmail.com](mailto:harryxiyou@gmail.com)>  
中文版翻譯者： 賈威威 Harry Wei <[harryxiyou@gmail.com](mailto:harryxiyou@gmail.com)>  
時奎亮 Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)>  
中文版校譯者： Yinglin Luan <[synmyth@gmail.com](mailto:synmyth@gmail.com)>  
Xiaochen Wang <[wangxiaochen0@gmail.com](mailto:wangxiaochen0@gmail.com)>  
yaxinsn <[yaxinsn@163.com](mailto:yaxinsn@163.com)>  
Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

## Linux 電郵客戶端配置信息

### Git

現在大多數開發人員使用 `git send-email` 而不是常規的電子郵件客戶端。這方面的手冊非常好。在接收端，維護人員使用 `git am` 加載補丁。

如果你是 git 新手，那麼把你的第一個補丁發送給你自己。將其保存為包含所有標題的原始文本。運行 `git am raw_email.txt`，然後使用 `git log` 查看更改日誌。如果工作正常，再將補丁發送到相應的郵件列表。

### 普通配置

Linux 內核補丁是通過郵件被提交的，最好把補丁作為郵件體的內嵌文本。有些維護者接收附件，但是附件的內容格式應該是”`text/plain`”。然而，附件一般是不贊成的，因為這會使補丁的引用部分在評論過程中變的很困難。

用來發送 Linux 內核補丁的郵件客戶端在發送補丁時應該處於文本的原始狀態。例如，他們不能改變或者刪除制表符或者空格，甚至是在每一行的開頭或者結尾。

不要通過”`format=flowed`” 模式發送補丁。這樣會引起不可預期以及有害的斷行。

不要讓你的郵件客戶端進行自動換行。這樣也會破壞你的補丁。

郵件客戶端不能改變文本的字符集編碼方式。要發送的補丁只能是 ASCII 或者 UTF-8 編碼方式，如果你使用 UTF-8 編碼方式發送郵件，那麼你將會避免一些可能發生的字符集問題。

郵件客戶端應該形成並且保持 `References:` 或者 `In-Reply-To:` 標題，那麼郵件話題就不會中斷。

複製粘帖 (或者剪貼粘帖) 通常不能用於補丁，因為制表符會轉換為空格。使用 `xclipboard`, `xclip` 或者 `xcutsel` 也許可以，但是最好測試一下或者避免使用複製粘帖。

不要在使用 PGP/GPG 署名的郵件中包含補丁。這樣會使得很多腳本不能讀取和適用於你的補丁。(這個問題應該是可以修復的)

在給內核郵件列表發送補丁之前，給自己發送一個補丁是個不錯的主意，保存接收到的郵件，將補丁用‘patch’命令打上，如果成功了，再給內核郵件列表發送。

## 一些郵件客戶端提示

這裡給出一些詳細的 MUA 配置提示，可以用於給 Linux 內核發送補丁。這些並不意味是所有的軟體包配置總結。

說明：TUI = 以文本為基礎的用戶接口 GUI = 圖形界面用戶接口

### Alpine (TUI)

配置選項：在” Sending Preferences” 部分：

- “Do Not Send Flowed Text” 必須開啓
- “Strip Whitespace Before Sending” 必須關閉

當寫郵件時，光標應該放在補丁會出現的地方，然後按下 CTRL-R 組合鍵，使指定的補丁文件嵌入到郵件中。

### Evolution (GUI)

一些開發者成功的使用它發送補丁

當選擇郵件選項：**Preformat** 從 Format->Heading->Preformatted (Ctrl-7) 或者工具欄

然後使用： Insert->Text File…(Alt-n x) 插入補丁文件。

你還可以” diff -Nru old.c new.c | xclip”，選擇 Preformat，然後使用中間鍵進行粘帖。

### Kmail (GUI)

一些開發者成功的使用它發送補丁。

默認設置不為 HTML 格式是合適的；不要啓用它。

當書寫一封郵件的時候，在選項下面不要選擇自動換行。唯一的缺點就是你在郵件中輸入的任何文本都不會被自動換行，因此你必須在發送補丁之前手動換行。最簡單的方法就是啓用自動換行來書寫郵件，然後把它保存為草稿。一旦你在草稿中再次打開它，它已經全部自動換行了，那麼你的郵件雖然沒有選擇自動換行，但是還不會失去已有的自動換行。

在郵件的底部，插入補丁之前，放上常用的補丁定界符：三個連字號 (—)。

然後在”Message”菜單條目，選擇插入文件，接著選取你的補丁文件。還有一個額外的選項，你可以通過它配置你的郵件建立工具欄菜單，還可以帶上”insert file”圖標。

你可以安全地通過 GPG 標記附件，但是內嵌補丁最好不要使用 GPG 標記它們。作為內嵌文本的簽發補丁，當從 GPG 中提取 7 位編碼時會使他們變的更加複雜。

如果你非要以附件的形式發送補丁，那麼就右鍵點擊附件，然後選中屬性，突出”Suggest automatic display”，這樣內嵌附件更容易讓讀者看到。

當你要保存將要發送的內嵌文本補丁，你可以從消息列表窗格選擇包含補丁的郵件，然後右擊選擇“save as”。你可以使用一個沒有更改的包含補丁的郵件，如果它是以正確的形式組成。當你正真在它自己的窗口之下察看，那時沒有選項可以保存郵件-已經有一個這樣的 bug 被匯報到了 kmail 的 bugzilla 並且希望這將會被處理。郵件是以只針對某個用戶可讀寫的權限被保存的，所以如果你想把郵件複製到其他地方，你不得不把他們的權限改為組或者整體可讀。

### Lotus Notes (GUI)

不要使用它。

### Mutt (TUI)

很多 Linux 開發人員使用 mutt 客戶端，所以證明它肯定工作的非常漂亮。

Mutt 不自帶編輯器，所以不管你使用什麼編輯器都不應該帶有自動斷行。大多數編輯器都帶有一個”insert file”選項，它可以通過不改變文件內容的方式插入文件。

‘vim’ 作為 mutt 的編輯器： set editor=“vi”

如果使用 xclip，敲入以下命令:set paste 按中鍵之前或者 shift-insert 或者使用:r filename 如果想要把補丁作為內嵌文本。(a)ttach 工作的很好，不帶有”set paste”。

你可以通過 git format-patch 生成補丁，然後用 Mutt 發送它們：

```
$ mutt -H 0001-some-bug-fix.patch
```

配置選項：它應該以默認設置的形式工作。然而，把”send\_charset”設置為”us-ascii::utf-8”也是一個不錯的主意。

Mutt 是高度可配置的。這裡是個使用 mutt 通過 Gmail 發送的補丁的最小配置：

```
# .muttrc
# ====== IMAP ======
set imap_user = 'yourusername@gmail.com'
set imap_pass = 'yourpassword'
set spoolfile = imaps://imap.gmail.com/INBOX
set folder = imaps://imap.gmail.com/
set record="imaps://imap.gmail.com/[Gmail]/Sent Mail"
set postponed="imaps://imap.gmail.com/[Gmail]/Drafts"
```

```

set mbox="imaps://imap.gmail.com/[Gmail]/All Mail"

# ===== SMTP =====
set smtp_url = "smtp://username@smtp.gmail.com:587/"
set smtp_pass = $imap_pass
set ssl_force_tls = yes # Require encrypted connection

# ===== Composition =====
set editor = `echo \$EDITOR`
set edit_headers = yes # See the headers when editing
set charset = UTF-8      # value of $LANG; also fallback for send_charset
# Sender, email address, and sign-off line must match
unset use_domain        # because joe@localhost is just embarrassing
set realname = "YOUR NAME"
set from = "username@gmail.com"
set use_from = yes

```

Mutt 文檔含有更多信息：

<http://dev.mutt.org/trac/wiki/UseCases/Gmail>

<http://dev.mutt.org/doc/manual.html>

## Pine (TUI)

Pine 過去有一些空格刪減問題，但是這些現在應該都被修復了。

如果可以，請使用 alpine(pine 的繼承者)

配置選項：- 最近的版本需要消除流程文本 - “no-strip-whitespace-before-send” 選項也是需要的。

## Sylpheed (GUI)

- 內嵌文本可以很好的工作（或者使用附件）。
- 允許使用外部的編輯器。
- 對於目錄較多時非常慢。
- 如果通過 non-SSL 連接，無法使用 TLS SMTP 授權。
- 在組成窗口中有一個很有用的 ruler bar。
- 紿地址本中添加地址就不會正確的了解顯示名。

### Thunderbird (GUI)

默認情況下，thunderbird 很容易損壞文本，但是還有一些方法可以強制它變得更好。

- 在用戶帳號設置里，組成和尋址，不要選擇” Compose messages in HTML format”。
- 編輯你的 Thunderbird 配置設置來使它不要拆行使用：user\_pref( “mailnews.wraplength” , 0);
- 編輯你的 Thunderbird 配置設置，使它不要使用” format=flowed” 格式：user\_pref( “mailnews.send\_plaintext\_flowed” , false);
- 你需要使 Thunderbird 變為預先格式方式：如果默認情況下你書寫的是 HTML 格式，那不是很難。僅僅從標題欄的下拉框中選擇” Preformat” 格式。如果默認情況下你書寫的是文本格式，你不得把它改為 HTML 格式（僅僅作為一次性的）來書寫新的消息，然後強制使它回到文本格式，否則它就會拆行。要實現它，在寫信的圖標上使用 shift 鍵來使它變為 HTML 格式，然後標題欄的下拉框中選擇” Preformat” 格式。
- 允許使用外部的編輯器：針對 Thunderbird 打補丁最簡單的方法就是使用一個” external editor” 擴展，然後使用你最喜歡的 \$EDITOR 來讀取或者合併補丁到文本中。要實現它，可以下載並且安裝這個擴展，然後添加一個使用它的按鍵 View->Toolbars->Customize…最後當你書寫信息的時候僅僅點擊它就可以了。

### TkRat (GUI)

可以使用它。使用” Insert file…” 或者外部的編輯器。

### Gmail (Web GUI)

不要使用它發送補丁。

Gmail 網頁客戶端自動地把制表符轉換為空格。

雖然制表符轉換為空格問題可以被外部編輯器解決，同時它還會使用回車換行把每行拆分為 78 個字符。

另一個問題是 Gmail 還會把任何不是 ASCII 的字符的信息改為 base64 編碼。它把東西變的像歐洲人的名字。

###

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

---

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件

給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:src.res@email.cn)。

**Original** Documentation/process/license-rules.rst

**Translator** Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)> Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

## Linux 內核許可規則

Linux 內核根據 LICENSES/pREFERRED/GPL-2.0 中提供的 GNU 通用公共許可證版本 2 (GPL-2.0) 的條款提供，並在 LICENSES/EXCEPTIONS/Linux-syscall-note 中顯式描述了例外的系統調用，如 COPYING 文件中所述。

此文檔文件提供了如何對每個源文件進行注釋以使其許可證清晰明確的說明。它不會取代內核的許可證。

內核原始碼作為一個整體適用於 COPYING 文件中描述的許可證，但是單個源文件可以具有不同的與 GPL-2.0 兼容的許可證：

GPL-1.0+ : GNU 通用公共許可證 v1.0 或更高版本

GPL-2.0+ : GNU 通用公共許可證 v2.0 或更高版本

LGPL-2.0 : 僅限 GNU 庫通用公共許可證 v2

LGPL-2.0+ : GNU 庫通用公共許可證 v2 或更高版本

LGPL-2.1 : 僅限 GNU 寬通用公共許可證 v2.1

LGPL-2.1+ : GNU 寬通用公共許可證 v2.1 或更高版本

除此之外，個人文件可以在雙重許可下提供，例如一個兼容的 GPL 變體，或者 BSD，MIT 等許可。

用戶空間 API (UAPI) 頭文件描述了用戶空間程序與內核的接口，這是一種特殊情況。根據內核 COPYING 文件中的注釋，syscall 接口是一個明確的邊界，它不會將 GPL 要求擴展到任何使用它與內核通信的軟體。由於 UAPI 頭文件必須包含在創建在 Linux 內核上運行的可執行文件的任何源文件中，因此此例外必須記錄在特別的許可證表達式中。

表達源文件許可證的常用方法是將匹配的樣板文本添加到文件的頂部注釋中。由於格式，拼寫錯誤等，這些「樣板」很難通過那些在上下文中使用的驗證許可證合規性的工具。

樣板文本的替代方法是在每個源文件中使用軟體包數據交換 (SPDX) 許可證標識符。SPDX 許可證標識符是機器可解析的，並且是用於提供文件內容的許可證的精確縮寫。SPDX 許可證標識符由 Linux 基金會的 SPDX 工作組管理，並得到了整個行業，工具供應商和法律團隊的合作夥伴的一致同意。有關詳細信息，請參閱 <https://spdx.org/>

Linux 內核需要所有源文件中的精確 SPDX 標識符。內核中使用的有效標識符在 [許可標識符](#) 一節中進行了解釋，並且已可以在 <https://spdx.org/licenses/> 上的官方 SPDX 許可證列表中檢索，並附帶許可證文本。

## 許可標識符語法

### 1. 安置:

內核文件中的 SPDX 許可證標識符應添加到可包含注釋的文件中的第一行。對於大多 數文件，這是第一行，除了那些在第一行中需要'#!PATH\_TO\_INTERPRETER'的腳本。對於這些腳本，SPDX 標識符進入第二行。

### 2. 風格:

SPDX 許可證標識符以注釋的形式添加。注釋樣式取決於文件類型:

```
C source: // SPDX-License-Identifier: <SPDX License Expression>
C header: /* SPDX-License-Identifier: <SPDX License Expression> */
ASM:      /* SPDX-License-Identifier: <SPDX License Expression> */
scripts:  # SPDX-License-Identifier: <SPDX License Expression>
.rst:     .. SPDX-License-Identifier: <SPDX License Expression>
.dts{i}: // SPDX-License-Identifier: <SPDX License Expression>
```

如果特定工具無法處理標準注釋樣式，則應使用工具接受的相應注釋機制。這是在 C 頭文件中使用「`/**/`」樣式注釋的原因。過去在使用生成的.lds 文件中觀察到構建被破壞，其中'ld' 無法解析 C++ 注釋。現在已經解決了這個問題，但仍然有較舊的彙編程序工具無法處理 C++ 樣式的注釋。

### 3. 句法:

`<SPDX 許可證表達式 >` 是 SPDX 許可證列表中的 SPDX 短格式許可證標識符，或者在許可證例外適用時由「WITH」分隔的兩個 SPDX 短格式許可證標識符的組合。當應用多個許可證時，表達式由分隔子表達式的關鍵字「AND」，「OR」組成，並由「(」，「)」包圍。

帶有「或更高」選項的 [L]GPL 等許可證的許可證標識符通過使用「+」來表示「或更高」選項來構建。：

```
// SPDX-License-Identifier: GPL-2.0+
// SPDX-License-Identifier: LGPL-2.1+
```

當需要修正的許可證時，應使用 WITH。 例如，linux 內核 UAPI 文件使用表達式:

```
// SPDX-License-Identifier: GPL-2.0 WITH Linux-syscall-note
// SPDX-License-Identifier: GPL-2.0+ WITH Linux-syscall-note
```

其它在內核中使用 WITH 例外的事例如下:

```
// SPDX-License-Identifier: GPL-2.0 WITH mif-exception
// SPDX-License-Identifier: GPL-2.0+ WITH GCC-exception-2.0
```

例外只能與特定的許可證標識符一起使用。有效的許可證標識符列在異常文本文件的標記中。有關詳細信息，請參閱[許可標識符](#)一章中的[例外](#)。

如果文件是雙重許可且只選擇一個許可證，則應使用 OR。例如，一些 dtsi 文件在雙許可下可用：

```
// SPDX-License-Identifier: GPL-2.0 OR BSD-3-Clause
```

內核中雙許可文件中許可表達式的示例：

```
// SPDX-License-Identifier: GPL-2.0 OR MIT
// SPDX-License-Identifier: GPL-2.0 OR BSD-2-Clause
// SPDX-License-Identifier: GPL-2.0 OR Apache-2.0
// SPDX-License-Identifier: GPL-2.0 OR MPL-1.1
// SPDX-License-Identifier: (GPL-2.0 WITH Linux-syscall-note) OR MIT
// SPDX-License-Identifier: GPL-1.0+ OR BSD-3-Clause OR OpenSSL
```

如果文件具有多個許可證，其條款全部適用於使用該文件，則應使用 AND。例如，如果代碼是從另一個項目繼承的，並且已經授予了將其放入內核的權限，但原始許可條款需要保持有效：

```
// SPDX-License-Identifier: (GPL-2.0 WITH Linux-syscall-note) AND MIT
```

另一個需要遵守兩套許可條款的例子是：

```
// SPDX-License-Identifier: GPL-1.0+ AND LGPL-2.1+
```

## 許可標識符

當前使用的許可證以及添加到內核的代碼許可證可以分解為：

### 1. 優先許可：

應儘可能使用這些許可證，因為它們已知完全兼容並廣泛使用。這些許可證在內核目錄：

```
LICENSES/pREFERRED/
```

此目錄中的文件包含完整的許可證文本和元標記。文件名與 SPDX 許可證標識符相同，後者應用於源文件中的許可證。

例如：

```
LICENSES/pREFERRED/GPL-2.0
```

包含 GPLv2 許可證文本和所需的元標籤：

### LICENSES/preferred/MIT

包含 MIT 許可證文本和所需的元標記

元標記:

許可證文件中必須包含以下元標記：

- Valid-License-Identifier:

一行或多行，聲明那些許可標識符在項目內有效，以引用此特定許可的文本。通常這是一個有效的標識符，但是例如對於帶有‘或更高’選項的許可證，兩個標識符都有效。

- SPDX-URL:

SPDX 頁面的 URL，其中包含與許可證相關的其他信息。

- Usage-Guidance:

使用建議的自由格式文本。該文本必須包含 SPDX 許可證標識符的正確示例，因為它們應根據[許可標識符語法](#)指南放入源文件中。

- License-Text:

此標記之後的所有文本都被視為原始許可文本

文件格式示例:

```
Valid-License-Identifier: GPL-2.0
```

```
Valid-License-Identifier: GPL-2.0+
```

```
SPDX-URL: https://spdx.org/licenses/GPL-2.0.html
```

Usage-Guide:

To use this license in source code, put one of the following SPDX tag/value pairs into a comment according to the placement guidelines in the licensing rules documentation.

For 'GNU General Public License (GPL) version 2 only' use:

```
SPDX-License-Identifier: GPL-2.0
```

For 'GNU General Public License (GPL) version 2 or any later version' use:

```
SPDX-License-Identifier: GPL-2.0+
```

License-Text:

```
Full license text
```

```
SPDX-License-Identifier: MIT
```

```
SPDX-URL: https://spdx.org/licenses/MIT.html
```

Usage-Guide:

To use this license in source code, put the following SPDX tag/value pair into a comment according to the placement guidelines in the licensing rules documentation.

```
SPDX-License-Identifier: MIT
```

License-Text:

```
Full license text
```

## 2. 不推薦的許可證:

這些許可證只應用於現有代碼或從其他項目導入代碼。這些許可證在內核目錄:

LICENSES/other/

此目錄中的文件包含完整的許可證文本和元標記。文件名與 SPDX 許可證標識符相同，後者應用於源文件中的許可證。

例如:

LICENSES/other/ISC

包含國際系統聯合許可文本和所需的元標籤:

LICENSES/other/ZLib

包含 ZLIB 許可文本和所需的元標籤.

元標籤:

「其他」許可證的元標籤要求與優先許可的要求相同。

文件格式示例:

Valid-License-Identifier: ISC

SPDX-URL: <https://spdx.org/licenses/ISC.html>

Usage-Guide:

Usage of this license in the kernel for new code is discouraged and it should solely be used for importing code from an already existing project.

To use this license in source code, put the following SPDX tag/value pair into a comment according to the placement guidelines in the licensing rules documentation.

SPDX-License-Identifier: ISC

License-Text:

Full license text

## 3. 例外:

某些許可證可以修改，並允許原始許可證不具有的某些例外權利。這些例外在內核目錄:

LICENSES/exceptions/

此目錄中的文件包含完整的例外文本和所需的例外元標記。

例如：

LICENSES/exceptions/Linux-syscall-note

包含 Linux 內核的 COPYING 文件中記錄的 Linux 系統調用例外，該文件用於 UAPI 頭文件。例如：

LICENSES/exceptions/GCC-exception-2.0

包含 GCC' 連結例外'，它允許獨立於其許可證的任何二進位文件與標記有此例外的文件的編譯版本連結。這是從 GPL 不兼容原始碼創建可運行的可執行文件所必需的。

例外元標記：

以下元標記必須在例外文件中可用：

- SPDX-Exception-Identifier:

一個可與 SPDX 許可證標識符一起使用的例外標識符。

- SPDX-URL:

SPDX 頁面的 URL，其中包含與例外相關的其他信息。

- SPDX-Licenses:

以逗號分隔的例外可用的 SPDX 許可證標識符列表。

- Usage-Guidance:

使用建議的自由格式文本。必須在文本後面加上 SPDX 許可證標識符的正確示例，因為它們應根據許可標識符語法 指南放入源文件中。

- Exception-Text:

此標記之後的所有文本都被視為原始異常文本

文件格式示例：

```
SPDX-Exception-Identifier: Linux-syscall-note
SPDX-URL: https://spdx.org/licenses/Linux-syscall-note.html
SPDX-Licenses: GPL-2.0, GPL-2.0+, GPL-1.0+, LGPL-2.0, LGPL-2.0+, LGPL-2.1, LGPL-2.1+
Usage-Guidance:
This exception is used together with one of the above SPDX-Licenses
to mark user-space API (uapi) header files so they can be included
into non GPL compliant user-space application code.
To use this exception add it with the keyword WITH to one of the
identifiers in the SPDX-Licenses tag:
SPDX-License-Identifier: <SPDX-License> WITH Linux-syscall-note
Exception-Text:
Full exception text
```

SPDX-Exception-Identifier: GCC-exception-2.0  
 SPDX-URL: <https://spdx.org/licenses/GCC-exception-2.0.html>  
 SPDX-Licenses: GPL-2.0, GPL-2.0+  
 Usage-Guidance:

The "GCC Runtime Library exception 2.0" is used together with one of the above SPDX-Licenses for code imported from the GCC runtime library.

To use this exception add it with the keyword WITH to one of the identifiers in the SPDX-Licenses tag:

SPDX-License-Identifier: <SPDX-License> WITH GCC-exception-2.0

Exception-Text:

Full exception text

所有 SPDX 許可證標識符和例外都必須在 LICENSES 子目錄中具有相應的文件。這是允許工具驗證（例如 `checkpatch.pl`）以及準備好從源讀取和提取許可證所必需的，這是各種 FOSS 組織推薦的，例如 FSFE REUSE initiative。

## 模塊許可

可加載內核模塊還需要 MODULE\_LICENSE () 標記。此標記既不替代正確的原始碼許可證信息 (SPDX-License-Identifier)，也不以任何方式表示或確定提供模塊原始碼的確切許可證。

此標記的唯一目的是提供足夠的信息，該模塊是否是自由軟體或者是內核模塊加載器和用戶空間工具的專有模塊。

MODULE\_LICENSE () 的有效許可證字符串是：

“GPL”	模塊是根據 GPL 版本 2 許可的。這並不表示僅限於 GPL-2.0 或 GPL-2.0 或更高版本之間的任何區別。最正確許可證信息只能通過相應源文件中的許可證信息來確定
“GPL v2”	和“GPL”相同，它的存在是因為歷史原因。
“GPL and additional rights”	表示模塊源在 GPL v2 變體和 MIT 許可下雙重許可的歷史變體。請不要在新代碼中使用。
“Dual MIT/GPL”	表達該模塊在 GPL v2 變體或 MIT 許可證選擇下雙重許可的正確方式。
“Dual BSD/GPL”	該模塊根據 GPL v2 變體或 BSD 許可證選擇進行雙重許可。BSD 許可證的確切變體只能通過相應源文件中的許可證信息來確定。
“Dual MPL/GPL”	該模塊根據 GPL v2 變體或 Mozilla Public License (MPL) 選項進行雙重許可。MPL 許可證的確切變體只能通過相應的源文件中的許可證信息來確定。
“Proprietary”	該模塊屬於專有許可。此字符串僅用於專有的第三方模塊，不能用於在內核樹中具有原始碼的模塊。以這種方式標記的模塊在加載時會使用‘P’標記汙染內核，並且內核模塊加載器拒絕將這些模塊連結到使用 EXPORT_SYMBOL_GPL() 導出的符號。

.. \_tw\_process\_statement\_kernel:

### orphan

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

**Original** Documentation/process/kernel-enforcement-statement.rst

**Translator** Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)> Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

## Linux 內核執行聲明

作為 Linux 內核的開發人員，我們對如何使用我們的軟體以及如何實施軟體許可證有著濃厚的興趣。遵守 GPL-2.0 的互惠共享義務對我們軟體和社區的長期可持續性至關重要。

雖然有權強制執行對我們社區的貢獻中的單獨版權權益，但我們有共同的利益，即確保個人強制執行行動的方式有利於我們的社區，不會對我們軟體生態系統的健康和增長產生意外的負面影響。為了阻止無益的執法行動，我們同意代表我們自己和我們版權利益的任何繼承人對 Linux 內核用戶作出以下符合我們開發社區最大利益的承諾：

儘管有 GPL-2.0 的終止條款，我們同意，採用以下 GPL-3.0 條款作為我們許可證下的附加許可，作為任何對許可證下權利的非防禦性主張，這符合我們開發社區的最佳利益。

但是，如果您停止所有違反本許可證的行為，則您從特定版權持有人處獲得的許可證將被恢復：(a) 暫時恢復，除非版權持有人明確並最終終止您的許可證；以及 (b) 永久恢復，如果版權持有人未能在您終止違反後 60 天內以合理方式通知您違反本許可證的行為，則永久恢復您的許可證。

此外，如果版權所有者以某種合理的方式通知您違反了本許可，這是您第一次從該版權所有者處收到違反本許可的通知（對於任何作品），並且您在收到通知後的 30 天內糾正違規行為。則您從特定版權所有者處獲得的許可將永久恢復。

我們提供這些保證的目的是鼓勵更多地使用該軟體。我們希望公司和個人使用、修改和分發此軟體。我們希望以公開和透明的方式與用戶合作，以消除我們對法規遵從性或強制執行的任何不確定性，這些不確定性可能會限制我們軟體的採用。我們將法律行動視為最後手段，只有在其他社區努力未能解決這一問題時才採取行動。

最後，一旦一個不合規問題得到解決，我們希望用戶會感到歡迎，加入我們為之努力的這個項目。共同努力，我們會更強大。

除了下面提到的以外，我們只為自己說話，而不是為今天、過去或將來可能為之工作的任何公司說話。

- Laura Abbott
- Bjorn Andersson (Linaro)
- Andrea Arcangeli
- Neil Armstrong
- Jens Axboe
- Pablo Neira Ayuso
- Khalid Aziz
- Ralf Baechle
- Felipe Balbi
- Arnd Bergmann
- Ard Biesheuvel

- Tim Bird
- Paolo Bonzini
- Christian Borntraeger
- Mark Brown (Linaro)
- Paul Burton
- Javier Martinez Canillas
- Rob Clark
- Kees Cook (Google)
- Jonathan Corbet
- Dennis Dalessandro
- Vivien Didelot (Savoir-faire Linux)
- Hans de Goede
- Mel Gorman (SUSE)
- Sven Eckelmann
- Alex Elder (Linaro)
- Fabio Estevam
- Larry Finger
- Bhumika Goyal
- Andy Gross
- Juergen Gross
- Shawn Guo
- Ulf Hansson
- Stephen Hemminger (Microsoft)
- Tejun Heo
- Rob Herring
- Masami Hiramatsu
- Michal Hocko
- Simon Hormann
- Johan Hovold (Hovold Consulting AB)
- Christophe JAILLET

- Olof Johansson
- Lee Jones (Linaro)
- Heiner Kallweit
- Srinivas Kandagatla
- Jan Kara
- Shuah Khan (Samsung)
- David Kershner
- Jaegeuk Kim
- Namhyung Kim
- Colin Ian King
- Jeff Kirsher
- Greg Kroah-Hartman (Linux Foundation)
- Christian König
- Vinod Koul
- Krzysztof Kozlowski
- Viresh Kumar
- Aneesh Kumar K.V
- Julia Lawall
- Doug Ledford
- Chuck Lever (Oracle)
- Daniel Lezcano
- Shaohua Li
- Xin Long
- Tony Luck
- Catalin Marinas (Arm Ltd)
- Mike Marshall
- Chris Mason
- Paul E. McKenney
- Arnaldo Carvalho de Melo
- David S. Miller

- Ingo Molnar
- Kuninori Morimoto
- Trond Myklebust
- Martin K. Petersen (Oracle)
- Borislav Petkov
- Jiri Pirko
- Josh Poimboeuf
- Sebastian Reichel (Collabora)
- Guenter Roeck
- Joerg Roedel
- Leon Romanovsky
- Steven Rostedt (VMware)
- Frank Rowand
- Ivan Safonov
- Anna Schumaker
- Jes Sorensen
- K.Y. Srinivasan
- David Sterba (SUSE)
- Heiko Stuebner
- Jiri Kosina (SUSE)
- Willy Tarreau
- Dmitry Torokhov
- Linus Torvalds
- Thierry Reding
- Rik van Riel
- Luis R. Rodriguez
- Geert Uytterhoeven (Glider bvba)
- Eduardo Valentin (Amazon.com)
- Daniel Vetter
- Linus Walleij

- Richard Weinberger
- Dan Williams
- Rafael J. Wysocki
- Arvind Yadav
- Masahiro Yamada
- Wei Yongjun
- Lv Zheng
- Marc Zyngier (Arm Ltd)

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:src.res@email.cn)。

**Original** Documentation/process/kernel-driver-statement.rst

**Translator** Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)> Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

## 內核驅動聲明

### 關於 Linux 內核模塊的立場聲明

我們，以下署名的 Linux 內核開發人員，認為任何封閉源 Linux 內核模塊或驅動程序都是有害的和不可取的。我們已經一再發現它們對 Linux 用戶，企業和更大的 Linux 生態系統有害。這樣的模塊否定了 Linux 開發模型的開放性，穩定性，靈活性和可維護性，並使他們的用戶無法使用 Linux 社區的專業知識。提供閉源內核模塊的供應商迫使其客戶放棄 Linux 的主要優勢或選擇新的供應商。因此，為了充分利用開源所提供的成本節省和共享支持優勢，我們敦促供應商採取措施，以開源內核代碼在 Linux 上為其客戶提供支持。

我們只為自己說話，而不是我們今天可能會為之工作，過去或將來會為之工作的任何公司。

- Dave Airlie
- Nick Andrew
- Jens Axboe
- Ralf Baechle
- Felipe Balbi

- Ohad Ben-Cohen
- Muli Ben-Yehuda
- Jiri Benc
- Arnd Bergmann
- Thomas Bogendoerfer
- Vitaly Bordug
- James Bottomley
- Josh Boyer
- Neil Brown
- Mark Brown
- David Brownell
- Michael Buesch
- Franck Bui-Huu
- Adrian Bunk
- François Cami
- Ralph Campbell
- Luiz Fernando N. Capitulino
- Mauro Carvalho Chehab
- Denis Cheng
- Jonathan Corbet
- Glauber Costa
- Alan Cox
- Magnus Damm
- Ahmed S. Darwish
- Robert P. J. Day
- Hans de Goede
- Arnaldo Carvalho de Melo
- Helge Deller
- Jean Delvare
- Mathieu Desnoyers

- Sven-Thorsten Dietrich
- Alexey Dobriyan
- Daniel Drake
- Alex Dubov
- Randy Dunlap
- Michael Ellerman
- Pekka Enberg
- Jan Engelhardt
- Mark Fasheh
- J. Bruce Fields
- Larry Finger
- Jeremy Fitzhardinge
- Mike Frysinger
- Kumar Gala
- Robin Getz
- Liam Girdwood
- Jan-Benedict Glaw
- Thomas Gleixner
- Brice Goglin
- Cyrill Gorcunov
- Andy Gospodarek
- Thomas Graf
- Krzysztof Halasa
- Harvey Harrison
- Stephen Hemminger
- Michael Hennerich
- Tejun Heo
- Benjamin Herrenschmidt
- Kristian Høgsberg
- Henrique de Moraes Holschuh

- Marcel Holtmann
- Mike Isely
- Takashi Iwai
- Olof Johansson
- Dave Jones
- Jesper Juhl
- Matthias Kaehlcke
- Kenji Kaneshige
- Jan Kara
- Jeremy Kerr
- Russell King
- Olaf Kirch
- Roel Kluin
- Hans-Jürgen Koch
- Auke Kok
- Peter Korsgaard
- Jiri Kosina
- Aaro Koskinen
- Mariusz Kozlowski
- Greg Kroah-Hartman
- Michael Krufky
- Aneesh Kumar
- Clemens Ladisch
- Christoph Lameter
- Gunnar Larisch
- Anders Larsen
- Grant Likely
- John W. Linville
- Yinghai Lu
- Tony Luck

- Pavel Macheck
- Matt Mackall
- Paul Mackerras
- Roland McGrath
- Patrick McHardy
- Kyle McMartin
- Paul Menage
- Thierry Merle
- Eric Miao
- Akinobu Mita
- Ingo Molnar
- James Morris
- Andrew Morton
- Paul Mundt
- Oleg Nesterov
- Luca Olivetti
- S.Çağlar Onur
- Pierre Ossman
- Keith Owens
- Venkatesh Pallipadi
- Nick Piggin
- Nicolas Pitre
- Evgeniy Polyakov
- Richard Purdie
- Mike Rapoport
- Sam Ravnborg
- Gerrit Renker
- Stefan Richter
- David Rientjes
- Luis R. Rodriguez

- Stefan Roesе
- Francois Romieu
- Rami Rosen
- Stephen Rothwell
- Maciej W. Rozycki
- Mark Salyzyn
- Yoshinori Sato
- Deepak Saxena
- Holger Schurig
- Amit Shah
- Yoshihiro Shimoda
- Sergei Shtylyov
- Kay Sievers
- Sebastian Siewior
- Rik Snel
- Jes Sorensen
- Alexey Starikovskiy
- Alan Stern
- Timur Tabi
- Hirokazu Takata
- Eliezer Tamir
- Eugene Teo
- Doug Thompson
- FUJITA Tomonori
- Dmitry Torokhov
- Marcelo Tosatti
- Steven Toth
- Theodore Tso
- Matthias Urlichhs
- Geert Uytterhoeven

- Arjan van de Ven
- Ivo van Doorn
- Rik van Riel
- Wim Van Sebroeck
- Hans Verkuil
- Horst H. von Brand
- Dmitri Vorobiev
- Anton Vorontsov
- Daniel Walker
- Johannes Weiner
- Harald Welte
- Matthew Wilcox
- Dan J. Williams
- Darrick J. Wong
- David Woodhouse
- Chris Wright
- Bryan Wu
- Rafael J. Wysocki
- Herbert Xu
- Vlad Yasevich
- Peter Zijlstra
- Bartłomiej Zolnierkiewicz

其它大多數開發人員感興趣的社區指南：

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

### Original Documentation/process/submitting-drivers.rst

如果想評論或更新本文的內容，請直接聯繫原文檔的維護者。如果你使用英文交流有困難的話，也可以向中文版維護者求助。如果本翻譯更新不及時或者翻譯存在問題，請聯繫中文版維護者：

中文版維護者： 李陽 Li Yang <leoyang.li@nxp.com>  
中文版翻譯者： 李陽 Li Yang <leoyang.li@nxp.com>  
中文版校譯者： 陳琦 Maggie Chen <chenqi@beyondsoft.com>  
                  王聰 Wang Cong <xifyou.wangcong@gmail.com>  
                  張巍 Zhang Wei <wezhang@outlook.com>  
                  胡皓文 Hu Haowen <src.res@email.cn>

## 如何向 Linux 內核提交驅動程序

這篇文章將會解釋如何向不同的內核源碼樹提交設備驅動程序。請注意，如果你感興趣的是顯卡驅動程序，你也許應該訪問 XFree86 項目 (<https://www.xfree86.org/>) 和／或 X.org 項目 (<https://x.org>)。

另請參閱 [如何讓你的改動進入內核](#) 文檔。

## 分配設備號

塊設備和字符設備的主設備號與從設備號是由 Linux 命名編號分配權威 LANANA（現在是 Torben Mathiasen）負責分配。申請的網址是 <https://www.lanana.org/>。即使不準備提交到主流內核的設備驅動也需要在這裡分配設備號。有關詳細信息，請參閱 Documentation/admin-guide/devices.rst。

如果你使用的不是已經分配的設備號，那麼當你提交設備驅動的時候，它將會被強制分配一個新的設備號，即便這個設備號和你之前發給客戶的截然不同。

## 設備驅動的提交對象

**Linux 2.0:** 此內核源碼樹不接受新的驅動程序。

**Linux 2.2:** 此內核源碼樹不接受新的驅動程序。

**Linux 2.4:** 如果所屬的代碼領域在內核的 MAINTAINERS 文件中列有一個總維護者，那麼請將驅動程序提交給他。如果此維護者沒有回應或者你找不到恰當的維護者，那麼請聯繫 Willy Tarreau <[w@1wt.eu](mailto:w@1wt.eu)>。

**Linux 2.6:** 除了遵循和 2.4 版內核同樣的規則外，你還需要在 linux-kernel 郵件列表上跟蹤最新的 API 變化。向 Linux 2.6 內核提交驅動的頂級聯繫人是 Andrew Morton <[akpm@linux-foundation.org](mailto:akpm@linux-foundation.org)>。

## 決定設備驅動能否被接受的條件

**許可：**代碼必須使用 **GNU 通用公開許可證 (GPL)** 提交給 **Linux**，但是 我們並不要求 **GPL** 是唯一的許可。你或許會希望同時使用多種許可證發布，如果希望驅動程序可以被其他開源社區（比如 **BSD**）使用。請參考 `include/linux/module.h` 文件中所列出的可被接受共存的許可。

**版權：**版權所有者必須同意使用 **GPL** 許可。最好提交者和版權所有者 是相同個人或實體。否則，必需列出授權使用 **GPL** 的版權所有人或實體，以備驗證之需。

**接口：**如果你的驅動程序使用現成的接口並且和其他同類的驅動程序行 為相似，而不是去發明無謂的新接口，那麼它將會更容易被接受。如果你需要一個 **Linux** 和 **NT** 的通用驅動接口，那麼請在用戶空間實現它。

**代碼：**請使用 **Documentation/process/coding-style.rst** 中所描述的 **Linux** 代碼風格。如果你的某些代碼段（例如那些與 **Windows** 驅動程序包共享的代碼段）需要使用其他格式，而你卻只希望維護一份代碼，那麼請將它們很好地區分出來，並且註明原因。

**可移植性：**請注意，指針並不永遠是 **32** 位的，不是所有的計算機都使用小 尾模式 (little endian) 存儲數據，不是所有的人都擁有浮點單元，不要隨便在你的驅動程序里嵌入 **x86** 彙編指令。只能在 **x86** 上運行的驅動程序一般是不受歡迎的。雖然你可能只有 **x86** 硬體，很難測試驅動程序在其他平台上是否可用，但是確保代碼可以被輕鬆地移植卻是很簡單的。

**清晰度：**做到所有人都能修補這個驅動程序將會很有好處，因為這樣你將 會直接收到修復的補丁而不是 bug 報告。如果你提交一個試圖隱藏硬體工作機理的驅動程序，那麼它將會被扔進廢紙簍。

**電源管理：**因為 **Linux** 正在被很多行動裝置和桌面系統使用，所以你的驅 動程序也很有可能被使用在這些設備上。它應該支持最基本的電源管理，即在需要的情況下實現系統級休眠和喚醒要用到的.`suspend` 和.`resume` 函數。你應該檢查你的驅動程序是否能正確地處理休眠與喚醒，如果實在無法確認，請至少把.`suspend` 函數定義成返回 -ENOSYS (功能未實現) 錯誤。你還應該嘗試確保你的驅動在什麼都不乾的情況下將耗電降到最低。要獲得驅動程序測試的指導，請參閱 `Documentation/power/drivers-testing.rst`。有關驅動程序電源管理問題相對全面的概述，請參閱 `Documentation/driver-api/pm/devices.rst`。

**管理：**如果一個驅動程序的作者還在進行有效的維護，那麼通常除了那 些明顯正確且不需要任何檢查的補丁以外，其他所有的補丁都會被轉發給作者。如果你希望成為驅動程序的聯繫人和更新者，最好在代碼注釋中寫明並且在 `MAINTAINERS` 文件中加入這個驅動程序的條目。

## 不影響設備驅動能否被接受的條件

**供應商：**由硬體供應商來維護驅動程序通常是一件好事。不過，如果源碼 樹里已經有其他人提供了可穩定工作的驅動程序，那麼請不要期望「我是供應商」會成為內核改用你的驅動程序的理由。理想的情況是：供應商與現有驅動程序的作者合作，構建一個統一完美的驅動程序。

**作者：**驅動程序是由大的 **Linux** 公司研發還是由你個人編寫，並不影 響其是否能被內核接受。沒有人對內核源碼樹享有特權。只要你充分了解內核社區，你就會發現這一點。

### 資源列表

**Linux 內核主源碼樹：** <ftp://kernel.org/pub/linux/kernel/> … == 你的國家代碼，例如“cn”、“us”、“uk”、“fr” 等等

**Linux 內核郵件列表：** [linux-kernel@vger.kernel.org](mailto:linux-kernel@vger.kernel.org) [可通過向 [majordomo@vger.kernel.org](mailto:majordomo@vger.kernel.org) 發郵件來訂閱]

**Linux 設備驅動程序，第三版（探討 2.6.10 版內核）：** <https://lwn.net/Kernel/LDD3/> （免費版）

**LWN.net:** 每周內核開發活動摘要 - <https://lwn.net/>

2.6 版中 API 的變更：

<https://lwn.net/Articles/2.6-kernel-api/>

將舊版內核的驅動程序移植到 2.6 版：

<https://lwn.net/Articles/driver-porting/>

**內核新手 (KernelNewbies):** 為新的內核開發者提供文檔和幫助 <https://kernelnewbies.org/>

**Linux USB 項目：** <http://www.linux-usb.org/>

**寫內核驅動的「不要」(Arjan van de Ven 著)：** <http://www.fenrus.org/how-to-not-write-a-device-driver.pdf>

**內核清潔工 (Kernel Janitor):** <https://kernelnewbies.org/KernelJanitors>

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

---

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

---

**Original** Documentation/process/submit-checklist.rst

**Translator** Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)> Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

## Linux 內核補丁提交清單

如果開發人員希望看到他們的內核補丁提交更快地被接受，那麼他們應該做一些基本的事情。

這些都是在 [Documentation/translations/zh\\_TW/process/submitting-patches.rst](#) 和其他有關提交 Linux 內核補丁的文檔中提供的。

- 1) 如果使用工具，則包括定義/聲明該工具的文件。不要依賴於其他頭文件拉入您使用的頭文件。
- 2) 乾淨的編譯：
  - a) 使用適用或修改的 CONFIG 選項 =y、=m 和 =n。沒有 GCC 警告/錯誤，沒有連結器警告/錯誤。
  - b) 通過 allnoconfig、allmodconfig
  - c) 使用 0=builddir 時可以成功編譯
- 3) 通過使用本地交叉編譯工具或其他一些構建場在多個 CPU 體系結構上構建。
- 4) PPC64 是一種很好的交叉編譯檢查體系結構，因為它傾向於對 64 位的數使用無符號長整型。
- 5) 如下所述 [Documentation/translations/zh\\_TW/process/coding-style.rst](#). 檢查您的補丁是否為常規樣式。在提交 (`scripts/check_patch.pl`) 之前，使用補丁樣式檢查器檢查是否有輕微的衝突。您應該能夠處理您的補丁中存在的所有違規行為。
- 6) 任何新的或修改過的 CONFIG 選項都不會弄髒配置菜單，並默認為關閉，除非它們符合 [Documentation/kbuild/kconfig-language.rst](#) 中記錄的異常條件，菜單屬性：默認值。
- 7) 所有新的 kconfig 選項都有幫助文本。
- 8) 已仔細審查了相關的 Kconfig 組合。這很難用測試來糾正——腦力在這裡是有回報的。
- 9) 用 sparse 檢查乾淨。
- 10) 使用 `make checkstack` 和 `make namespacecheck` 並修復他們發現的任何問題。

---

**Note:** `checkstack` 並沒有明確指出問題，但是任何一個在堆棧上使用超過 512 字節的函數都可以進行更改。

---

- 11) 包括 kernel-doc 內核文檔以記錄全局內核 API。(靜態函數不需要，但也可以。) 使用 `make htmldocs` 或 `make pdfdocs` 檢查 kernel-doc 並修復任何問題。
- 12) 通過以下選項同時啓用的測試 CONFIG\_PREEMPT, CONFIG\_DEBUG\_PREEMPT, CONFIG\_DEBUG\_SLAB, CONFIG\_DEBUG\_PAGEALLOC, CONFIG\_DEBUG\_MUTEXES, CONFIG\_DEBUG\_SPINLOCK, CONFIG\_DEBUG\_ATOMIC\_SLEEP, CONFIG\_PROVE\_RCU 和 CONFIG\_DEBUG\_OBJECTS\_RCU\_HEAD
- 13) 已經過構建和運行時測試，包括有或沒有 CONFIG\_SMP, CONFIG\_PREEMPT.
- 14) 如果補丁程序影響 IO/磁碟等：使用或不使用 CONFIG\_LBDAF 進行測試。
- 15) 所有代碼路徑都已在啓用所有 lockdep 功能的情況下運行。
- 16) 所有新的/proc 條目都記錄在 [Documentation/](#)

- 17) 所有新的內核引導參數都記錄在 Documentation/admin-guide/kernel-parameters.rst 中。
- 18) 所有新的模塊參數都記錄在 MODULE\_PARM\_DESC()
- 19) 所有新的用戶空間接口都記錄在 Documentation/ABI/ 中。有關詳細信息，請參閱 Documentation/ABI/ README 。更改用戶空間接口的補丁應該抄送 [linux-api@vger.kernel.org](mailto:linux-api@vger.kernel.org) 。
- 20) 已通過至少注入 slab 和 page 分配失敗進行檢查。請參閱 Documentation/fault-injection/ 如果新代碼是實質性的，那麼添加子系統特定的故障注入可能是合適的。
- 21) 新添加的代碼已經用 gcc -W 編譯（使用 make EXTRA\_CFLAGS=-W ）。這將產生大量噪聲，但對於查找諸如「警告：有符號和無符號之間的比較」之類的錯誤很有用。
- 22) 在它被合併到-mm 補丁集中之後進行測試，以確保它仍然與所有其他排隊的補丁以及 VM 、 VFS 和其他子系統中的各種更改一起工作。
- 23) 所有內存屏障例如 barrier() , rmb() , wmb() 都需要原始碼中的注釋來解釋它們正在執行的操作及其原因的邏輯。
- 24) 如果補丁添加了任何 ioctl ，那麼也要更新 Documentation/userspace-api/ioctl/ioctl-number.rst
- 25) 如果修改後的原始碼依賴或使用與以下 Kconfig 符號相關的任何內核 API 或功能，則在禁用相關 Kconfig 符號和/or =m （如果該選項可用）的情況下測試以下多個構建 [ 並非所有這些都同時存在，只是它們的各種/隨機組合 ] :  
CONFIG\_SMP, CONFIG\_SYSFS, CONFIG\_PROC\_FS, CONFIG\_INPUT, CONFIG\_PCI, CONFIG\_BLOCK, CONFIG\_PM, CONFIG\_MAGIC\_SYSRQ, CONFIG\_NET, CONFIG\_INET=n (但是後者伴隨 CONFIG\_NET=y).

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>) 。

### Original Documentation/process/stable-api-nonsense.rst

譯者：

中文版維護者：鍾宇 TripleX Chung <xxx.phy@gmail.com>  
中文版翻譯者：鍾宇 TripleX Chung <xxx.phy@gmail.com>  
中文版校譯者：李陽 Li Yang <leoyang.li@nxp.com>  
胡皓文 Hu Haowen <src.res@email.cn>

## Linux 內核驅動接口

寫作本文檔的目的，是為了解釋為什麼 Linux 既沒有二進位內核接口，也沒有穩定的內核接口。這裡所說的內核接口，是指內核里的接口，而不是內核和用戶空間的接口。內核到用戶空間的接口，是提供給應用程式使用的系統調用，系統調用在歷史上幾乎沒有過變化，將來也不會有變化。我有一些老應用程式是在 0.9 版本或者更早版本的內核上編譯的，在使用 2.6 版本內核的 Linux 發布上依然用得很好。用戶和應用程式作者可以將這個接口看成是穩定的。

### 執行綱要

你也許以為自己想要穩定的內核接口，但是你不清楚你要的實際上不是它。你需要的其實是穩定的驅動程序，而你只有將驅動程序放到公版內核的原始碼樹里，才有可能達到這個目的。而且這樣做還有很多其它好處，正是因為這些好處使得 Linux 能成為強壯，穩定，成熟的作業系統，這也是你最開始選擇 Linux 的原因。

### 入門

只有那些寫驅動程序的「怪人」才會擔心內核接口的改變，對廣大用戶來說，既看不到內核接口，也不需要去關心它。

首先，我不打算討論關於任何非 GPL 許可的內核驅動的法律問題，這些非 GPL 許可的驅動程序包括不公開原始碼，隱藏原始碼，二進位或者是用原始碼包裝，或者是其它任何形式的不能以 GPL 許可公開原始碼的驅動程序。如果有法律問題，請諮詢律師，我只是一個程式設計師，所以我只打算探討技術問題（不是小看法律問題，法律問題很實際，並且需要一直關注）。

既然只談技術問題，我們就有了下面兩個主題：二進位內核接口和穩定的內核源代碼接口。這兩個問題是互相關聯的，讓我們先解決掉二進位接口的問題。

### 二進位內核接口

假如我們有一個穩定的內核原始碼接口，那麼自然而然的，我們就擁有了穩定的二進位接口，是這樣的嗎？錯。讓我們看看關於 Linux 內核的幾點事實：

- 取決於所用的 C 編譯器的版本，不同的內核數據結構里的結構體的對齊方式會有差別，代碼中不同函數的表現形式也不一樣（函數是不是被 `inline` 編譯取決於編譯器行爲）。不同的函數的表現形式並不重要，但是數據結構內部的對齊方式很關鍵。
- 取決於內核的配置選項，不同的選項會讓內核的很多東西發生改變：
  - 同一個結構體可能包含不同的成員變量
  - 有的函數可能根本不會被實現（比如編譯的時候沒有選擇 SMP 支持一些鎖函數就會被定義成空函數）。
  - 內核使用的內存會以不同的方式對齊，這取決於不同的內核配置選項。

- Linux 可以在很多的不同體系結構的處理器上運行。在某個體系結構上編譯好的二進位驅動程序，不可能在另外一個體系結構上正確的運行。

對於一個特定的內核，滿足這些條件並不難，使用同一個 C 編譯器和同樣的內核配置選項來編譯驅動程序模塊就可以了。這對於給一個特定 Linux 發布的特定版本提供驅動程序，是完全可以滿足需求的。但是如果你要給不同發布的不同版本都發布一個驅動程序，就需要在每個發布上用不同的內核設置參數都編譯一次內核，這簡直跟噩夢一樣。而且還要注意到，每個 Linux 發布還提供不同的 Linux 內核，這些內核都針對不同的硬體類型進行了優化（有很多種不同的處理器，還有不同的內核設置選項）。所以每發布一次驅動程序，都需要提供很多不同版本的內核模塊。

相信我，如果你真的要採取這種發布方式，一定會慢慢瘋掉，我很久以前就有過深刻的教訓…

### 穩定的內核原始碼接口

如果有人不將他的內核驅動程序，放入公版內核的原始碼樹，而又想讓驅動程序一直保持在最新的內核中可用，那麼這個話題將會變得沒完沒了。內核開發是持續而且快節奏的，從來都不會慢下來。內核開發人員在當前接口中找到 bug，或者找到更好的實現方式。一旦發現這些，他們就很快會去修改當前的接口。修改接口意味著，函數名可能會改變，結構體可能被擴充或者刪減，函數的參數也可能發生改變。一旦接口被修改，內核中使用這些接口的地方需要同時修正，這樣才能保證所有的東西繼續工作。

舉一個例子，內核的 USB 驅動程序接口在 USB 子系統的整個生命周期中，至少經歷了三次重寫。這些重寫解決以下問題：

- 把數據流從同步模式改成非同步模式，這個改動減少了一些驅動程序的複雜度，提高了所有 USB 驅動程序的吞吐率，這樣幾乎所有的 USB 設備都能以最大速率工作了。
- 修改了 USB 核心代碼中為 USB 驅動分配數據包內存的方式，所有的驅動都需要提供更多的參數給 USB 核心，以修正了很多已經被記錄在案的死鎖。

這和一些封閉原始碼的作業系統形成鮮明的對比，在那些作業系統上，不得不額外的維護舊的 USB 接口。這導致了一個可能性，新的開發者依然會不小心使用舊的接口，以不恰當的方式編寫代碼，進而影響到作業系統的穩定性。在上面的例子中，所有的開發者都同意這些重要的改動，在這樣的情況下修改代價很低。如果 Linux 保持一個穩定的內核原始碼接口，那麼就得創建一個新的接口；舊的，有問題的接口必須一直維護，給 Linux USB 開發者帶來額外的工作。既然所有的 Linux USB 驅動的作者都是利用自己的時間工作，那麼要求他們去做毫無意義的免費額外工作，是不可能的。安全問題對 Linux 來說十分重要。一個安全問題被發現，就會在短時間內得到修正。在很多情況下，這將導致 Linux 內核中的一些接口被重寫，以從根本上避免安全問題。一旦接口被重寫，所有使用這些接口的驅動程序，必須同時得到修正，以確定安全問題已經得到修復並且不可能在未來還有同樣的安全問題。如果內核內部接口不允許改變，那麼就不可能修復這樣的安全部問題，也不可能確認這樣的安全部問題以後不會發生。開發者一直在清理內核接口。如果一個接口沒有人在使用了，它就會被刪除。這樣可以確保內核儘可能的小，而且所有潛在的接口都會得到儘可能完整的測試（沒有人使用的接口是不可能得到良好的測試的）。

## 要做什么

如果你寫了一個 Linux 內核驅動，但是它還不在 Linux 原始碼樹里，作為一個開發者，你應該怎麼做？為每個發布的每個版本提供一個二進位驅動，那簡直是一個噩夢，要跟上永遠處於變化之中的內核接口，也是一件辛苦活。很簡單，讓你的驅動進入內核原始碼樹（要記得我們在談論的是以 GPL 許可發行的驅動，如果你的代碼不符合 GPL，那麼祝你好運，你只能自己解決這個問題了，你這個吸血鬼 < 把 Andrew 和 Linus 對吸血鬼的定義連結到這裡 >）。當你的代碼加入公版內核原始碼樹之後，如果一個內核接口改變，你的驅動會直接被修改接口的那個人修改。保證你的驅動永遠都可以編譯通過，並且一直工作，你幾乎不需要做什麼事情。

把驅動放到內核原始碼樹里會有很多的好處：

- 驅動的質量會提升，而維護成本（對原始作者來說）會下降。
- 其他人會給驅動添加新特性。
- 其他人會找到驅動中的 bug 並修復。
- 其他人會在驅動中找到性能優化的機會。
- 當外部的接口的改變需要修改驅動程序的時候，其他人會修改驅動程序
- 不需要聯繫任何發行商，這個驅動會自動的隨著所有的 Linux 發布一起發布。

和別的作業系統相比，Linux 為更多不同的設備提供現成的驅動，而且能在更多不同體系結構的處理器上支持這些設備。這個經過考驗的開發模式，必然是錯不了的:)

## 感謝

感謝 Randy Dunlap, Andrew Morton, David Brownell, Hanna Linder, Robert Love, and Nishanth Aravamudan 對於本文檔早期版本的評審和建議。

英文版維護者：Greg Kroah-Hartman <[greg@kroah.com](mailto:greg@kroah.com)>

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

---

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:src.res@email.cn)。

---

## Original Documentation/process/stable-kernel-rules.rst

如果想評論或更新本文的內容，請直接聯繫原文檔的維護者。如果你使用英文交流有困難的話，也可以向中文版維護者求助。如果本翻譯更新不及時或者翻譯存在問題，請聯繫中文版維護者：

中文版維護者：鍾宇 TripleX Chung <xxx.phy@gmail.com>

中文版翻譯者：鍾宇 TripleX Chung <xxx.phy@gmail.com>

中文版校譯者：

- 李陽 Li Yang <leoyang.li@nxp.com>
- Kangkai Yin <e12051@motorola.com>
- 胡皓文 Hu Haowen <src.res@email.cn>

### 所有你想知道的事情 - 關於 Linux 穩定版發布

關於 Linux 2.6 穩定版發布，所有你想知道的事情。

#### 關於哪些類型的補丁可以被接收進入穩定版代碼樹，哪些不可以的規則：

- 必須是顯而易見的正確，並且經過測試的。
- 連同上下文，不能大於 100 行。
- 必須只修正一件事情。
- 必須修正了一個給大家帶來麻煩的真正的 bug（不是「這也許是一個問題…」那樣的東西）。
- 必須修正帶來如下後果的問題：編譯錯誤（對被標記為 CONFIG\_BROKEN 的例外），內核崩潰，掛起，數據損壞，真正的安全問題，或者一些類似「哦，這不好」的問題。簡短的說，就是一些致命的問題。
- 沒有「理論上的競爭條件」，除非能給出競爭條件如何被利用的解釋。
- 不能存在任何的「瑣碎的」修正（拼寫修正，去掉多餘空格之類的）。
- 必須被相關子系統的維護者接受。
- 必須遵循 Documentation/translations/zh\_TW/process/submitting-patches.rst 里的規則。

#### 向穩定版代碼樹提交補丁的過程：

- 在確認了補丁符合以上的規則後，將補丁發送到 stable@vger.kernel.org。
- 如果補丁被接受到隊列里，發送者會收到一個 ACK 回復，如果沒有被接受，收到的是 NAK 回復。回復需要幾天的時間，這取決於開發者的時間安排。
- 被接受的補丁會被加到穩定版本隊列里，等待其他開發者的審查。
- 安全方面的補丁不要發到這個列表，應該發送到 security@kernel.org。

## 審查周期：

- 當穩定版的維護者決定開始一個審查周期，補丁將被發送到審查委員會，以及被補丁影響的領域的維護者（除非提交者就是該領域的維護者）並且抄送到 linux-kernel 郵件列表。
- 審查委員會有 48 小時的時間，用來決定給該補丁回復 ACK 還是 NAK。
- 如果委員會中有成員拒絕這個補丁，或者 linux-kernel 列表上有人反對這個補丁，並提出維護者和審查委員會之前沒有意識到的問題，補丁會從隊列中丟棄。
- 在審查周期結束的時候，那些得到 ACK 回應的補丁將會被加入到最新的穩定版發布中，一個新的穩定版發布就此產生。
- 安全性補丁將從內核安全小組那裡直接接收到穩定版代碼樹中，而不是通過通常的審查周期。請聯繫內核安全小組以獲得關於這個過程的更多細節。

## 審查委員會：

- 由一些自願承擔這項任務的內核開發者，和幾個非志願的組成。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

**Original** Documentation/process/management-style.rst

**Translator** Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)> Hu Haowen <[src.res@email.cn](mailto:<src.res@email.cn>)>

## Linux 內核管理風格

這是一個簡短的文檔，描述了 Linux 內核首選的（或胡編的，取決於您問誰）管理風格。它的目的是在某種程度上參照 process/coding-style.rst 主要是為了避免反覆回答<sup>1</sup> 相同（或類似）的問題。

管理風格是非常個人化的，比簡單的編碼風格規則更難以量化，因此本文檔可能與實際情況有關，也可能與實際情況無關。起初它是一個玩笑，但這並不意味著它可能不是真的。你得自己決定。

順便說一句，在談到「核心管理者」時，主要是技術負責人，而不是在公司內部進行傳統管理的人。如果你簽署了採購訂單或者對你的團隊的預算有任何了解，你幾乎肯定不是一個核心管理者。這些建議可能適用於您，也可能不適用於您。

<sup>1</sup> 本文件並不是通過回答問題，而是通過讓提問者痛苦地明白，我們不知道答案是什麼。

首先，我建議你購買「高效人的七個習慣」，而不是閱讀它。燒了它，這是一個偉大的象徵性姿態。

不管怎樣，這裡是：

### 1) 決策

每個人都認為管理者做決定，而且決策很重要。決定越大越痛苦，管理者就必須越高級。這很明顯，但事實並非如此。

遊戲的名字是 **避免做出決定**。尤其是，如果有人告訴你「選擇 (a) 或 (b)，我們真的需要你來做決定」，你就是陷入麻煩的管理者。你管理的人比你更了解細節，所以如果他們來找你做技術決策，你完蛋了。你顯然沒有能力為他們做這個決定。

(推論：如果你管理的人不比你更了解細節，你也會被搞砸，儘管原因完全不同。也就是說，你的工作是錯的，他們應該管理你的才智)

所以遊戲的名字是 **避免做出決定**，至少是那些大而痛苦的決定。做一些小的和非結果性的決定是很好的，並且使您看起來好像知道自己在做什麼，所以內核管理者需要做的是將那些大的和痛苦的決定變成那些沒有人真正關心的小事情。

這有助於認識到一個大的決定和一個小的決定之間的關鍵區別是你是否可以在事後修正你的決定。任何決定都可以通過始終確保如果你錯了（而且你一定會錯），你以後總是可以通過回溯來彌補損失。突然間，你就要做兩個無關緊要的決定，一個是錯誤的，另一個是正確的。

人們甚至會認為這是真正的領導能力（咳，胡說，咳）。

因此，避免重大決策的關鍵在於避免做那些無法挽回的事情。不要被引導到一個你無法逃離的角落。走投無路的老鼠可能很危險——走投無路的管理者真可憐。

事實證明，由於沒有人會愚蠢到讓內核管理者承擔巨大的財政責任，所以通常很容易回溯。既然你不可能浪費掉你無法償還的巨額資金，你唯一可以回溯的就是技術決策，而回溯很容易：只要告訴大家你是個不稱職的傻瓜，說對不起，然後撤銷你去年讓別人所做的毫無價值的工作。突然間，你一年前做的決定不在是一個重大的決定，因為它很容易被推翻。

事實證明，有些人對接受這種方法有困難，原因有兩個：

- 承認你是個白癡比看起來更難。我們都喜歡保持形象，在公共場合說你錯了有時確實很難。
- 如果有人告訴你，你去年所做的工作終究是不值得的，那麼對那些可憐的低級工程師來說也是很困難的，雖然實際的 **工作**很容易刪除，但你可能已經不可挽回地失去了工程師的信任。記住：「**不可撤銷**」是我們一開始就試圖避免的，而你的決定終究是一個重大的決定。

令人欣慰的是，這兩個原因都可以通過預先承認你沒有任何線索，提前告訴人們你的決定完全是初步的，而且可能是錯誤的事情來有效地緩解。你應該始終保留改變主意的權利，並讓人們 **意識到**這一點。當你 **還沒有**做過真正愚蠢的事情的時候，承認自己是愚蠢的要容易得多。

然後，當它真的被證明是愚蠢的時候，人們就轉動他們的眼珠說「哎呀，下次不要了」。

這種對不稱職的先發制人的承認，也可能使真正做這項工作的人也會三思是否值得做。畢竟，如果他們不確定這是否是一個好主意，你肯定不應該通過向他們保證他們所做的工作將會進入（內核）鼓勵他們。在他們開

始一項巨大的努力之前，至少讓他們三思而後行。

記住：他們最好比你更了解細節，而且他們通常認為他們對每件事都有答案。作為一個管理者，你能做的最好的事情不是灌輸自信，而是對他們所做的事情進行健康的批判性思考。

順便說一句，另一種避免做出決定的方法是看起來很可憐的抱怨「我們不能兩者兼得嗎？」相信我，它是有效的。如果不清楚哪種方法更好，他們最終會弄清楚的。最終的答案可能是兩個團隊都會因為這種情況而感到沮喪，以至於他們放棄了。

這聽起來像是一個失敗，但這通常是一個跡象，表明兩個項目都有問題，而參與其中的人不能做決定的原因是他們都是錯誤的。你最終會聞到玫瑰的味道，你避免了另一個你本可以搞砸的決定。

## 2) 人

大多數人都是白癡，做一名管理者意味著你必須處理好這件事，也許更重要的是，**他們**必須處理好你。

事實證明，雖然很容易糾正技術錯誤，但不容易糾正人格障礙。你只能和他們的和你的（人格障礙）共處。

但是，為了做好作為內核管理者的準備，最好記住不要燒掉任何橋樑，不要轟炸任何無辜的村民，也不要疏遠太多的內核開發人員。事實證明，疏遠人是相當容易的，而親近一個疏遠的人是很難的。因此，「疏遠」立即屬於「不可逆」的範疇，並根據1) 決策成為絕不可以做的事情。

這裡只有幾個簡單的規則：

- (1) 不要叫人笨蛋（至少不要在公共場合）
- (2) 學習如何在忘記規則(1)時道歉

問題在於 #1 很容易去做，因為你可以用數百萬種不同的方式說「你是一個笨蛋」<sup>2</sup> 有時甚至沒有意識到，而且幾乎總是帶著一種白熱化的信念，認為你是對的。

你越確信自己是對的（讓我們面對現實吧，你可以把幾乎所有人都稱為壞人，而且你經常是對的），事後道歉就越難。

要解決此問題，您實際上只有兩個選項：

- 非常擅長道歉
- 把「愛」均勻地散開，沒有人會真正感覺到自己被不公平地瞄準了。讓它有足夠的創造性，他們甚至可能會覺得好笑。

選擇永遠保持禮貌是不存在的。沒有人會相信一個如此明顯地隱藏了他們真實性格的人。

---

<sup>2</sup> 保羅·西蒙演唱了「離開愛人的 50 種方法」，因為坦率地說，「告訴開發者他們是 D\*CKHEAD」的 100 萬種方法都無法確認。但我確信他已經這麼想了。

### 3) 人 2 - 好人

雖然大多數人都是白癡，但不幸的是，據此推論你也是白癡，儘管我們都自我感覺良好，我們比普通人更好（讓我們面對現實吧，沒有人相信他們是普通人或低於普通人），我們也應該承認我們不是最鋒利的刀，而且會有其他人比你更不像白癡。

有些人對聰明人反應不好。其他人利用它們。

作為內核維護人員，確保您在第二組中。接受他們，因為他們會讓你的工作更容易。特別是，他們能夠為你做決定，這就是遊戲的全部內容。

所以當你發現一個比你聰明的人時，就順其自然吧。你的管理職責在很大程度上變成了「聽起來像是個好主意——去嘗試吧」，或者「聽起來不錯，但是 XXX 呢？」。「第二個版本尤其是一個很好的方法，要麼學習一些關於「XXX」的新東西，要麼通過指出一些聰明人沒有想到的東西來顯得更具管理性。無論哪種情況，你都會贏。

要注意的一件事是認識到一個領域的偉大不一定會轉化為其他領域。所以你可能會向特定的方向刺激人們，但讓我們面對現實吧，他們可能擅長他們所做的事情，而且對其他事情都很差勁。好消息是，人們往往會自然而然地重拾他們擅長的東西，所以當你向某個方向刺激他們時，你並不是在做不可逆轉的事情，只是不要用力推。

### 4) 責備

事情會出問題的，人們希望去責備人。貼標籤，你就是受責備的人。

事實上，接受責備並不難，尤其是當人們意識到這不全是你們的過錯時。這讓我們找到了承擔責任的最佳方式：為別人承擔這件事。你會感覺很好，他們會感覺很好，沒有受到指責。那誰，失去了他們的全部 36GB 色情收藏的人，因為你的無能將勉強承認，你至少沒有試圖逃避責任。

然後讓真正搞砸了的開發人員（如果你能找到他們）私下知道他們搞砸了。不僅是為了將來可以避免，而且為了讓他們知道他們欠你一個人情。而且，也許更重要的是，他們也可能是能夠解決問題的人。因為，讓我們面對現實吧，肯定不是你。

承擔責任也是你首先成為管理者的原因。這是讓人們信任你，讓你獲得潛在的榮耀的一部分，因為你就是那個會說「我搞砸了」的人。如果你已經遵循了以前的規則，你現在已經很擅長說了。

### 5) 應避免的事情

有一件事人們甚至比被稱為「笨蛋」更討厭，那就是在一個神聖的聲音中被稱為「笨蛋」。第一個你可以道歉，第二個你不會真正得到機會。即使你做得很好，他們也可能不再傾聽。

我們都認為自己比別人強，這意味著當別人裝腔作勢時，這會讓我們很惱火。你也許在道德和智力上比你周圍的每個人都優越，但不要試圖太明顯，除非你真的打算激怒某人<sup>3</sup>

<sup>3</sup> 提示：與你的工作沒有直接關係的網絡新聞組是消除你對他人不滿的好方法。偶爾寫些侮辱性的帖子，打個噴嚏，讓你的情緒得到淨化。別把牢騷帶回家

同樣，不要對事情太客氣或太微妙。禮貌很容易落得落花流水，把問題隱藏起來，正如他們所說，「在網際網路上，沒人能聽到你的含蓄。」用一個鈍器把這一點錘進去，因為你不能真的依靠別人來獲得你的觀點。

一些幽默可以幫助緩和直率和道德化。過度到荒謬的地步，可以灌輸一個觀點，而不會讓接受者感到痛苦，他們只是認為你是愚蠢的。因此，它可以幫助我們擺脫對批評的個人心理障礙。

## 6) 為什麼是我？

既然你的主要責任似乎是為別人的錯誤承擔責任，並且讓別人痛苦地明白你是不稱職的，那麼顯而易見的問題之一就變成了為什麼首先要這樣做。

首先，雖然你可能會或可能不會聽到十幾歲女孩（或男孩，讓我們不要在這裡評判或性別歧視）敲你的更衣室門，你會得到一個巨大的個人成就感為「負責」。別介意你真的在領導別人，你要跟上別人，儘可能快地追趕他們。每個人都會認為你是負責人。

如果你可以做到這個，這是個偉大的工作！

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

**Original** Documentation/process/embargoed-hardware-issues.rst

**Translator** Alex Shi <[alex.shi@linux.alibaba.com](mailto:alex.shi@linux.alibaba.com)> Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

## 被限制的硬體問題

### 範圍

導致安全問題的硬體問題與只影響 Linux 內核的純軟體錯誤是不同的安全錯誤類別。

必須區別對待諸如熔毀 (Meltdown)、Spectre、L1TF 等硬體問題，因為它們通常會影響所有作業系統（「OS」），因此需要在不同的 OS 供應商、發行版、硬體供應商和其他各方之間進行協調。對於某些問題，軟體緩解可能依賴於微碼或固件更新，這需要進一步的協調。

### 接觸

Linux 內核硬體安全小組獨立於普通的 Linux 內核安全小組。

該小組只負責協調被限制的硬體安全問題。Linux 內核中純軟體安全漏洞的報告不由該小組處理，報告者將被引導至常規 Linux 內核安全小組 (Documentation/admin-guide/) 聯繫。

可以通過電子郵件 <[hardware-security@kernel.org](mailto:hardware-security@kernel.org)> 與小組聯繫。這是一份私密的安全官名單，他們將幫助您根據我們的文檔化流程協調問題。

郵件列表是加密的，發送到列表的電子郵件可以通過 PGP 或 S/MIME 加密，並且必須使用報告者的 PGP 密鑰或 S/MIME 證書簽名。該列表的 PGP 密鑰和 S/MIME 證書可從 <https://www.kernel.org/>… 獲得。

雖然硬體安全問題通常由受影響的硬體供應商處理，但我們歡迎發現潛在硬體缺陷的研究人員或個人與我們聯繫。

### 硬體安全官

目前的硬體安全官小組:

- Linus Torvalds (Linux 基金會院士)
- Greg Kroah Hartman (Linux 基金會院士)
- Thomas Gleixner (Linux 基金會院士)

### 郵件列表的操作

處理流程中使用的加密郵件列表託管在 Linux Foundation 的 IT 基礎設施上。通過提供這項服務，Linux 基金會的 IT 基礎設施安全總監在技術上有能力訪問被限制的信息，但根據他的僱傭合同，他必須保密。Linux 基金會的 IT 基礎設施安全總監還負責 kernel.org 基礎設施。

Linux 基金會目前的 IT 基礎設施安全總監是 Konstantin Ryabitsev。

### 保密協議

Linux 內核硬體安全小組不是正式的機構，因此無法簽訂任何保密協議。核心社區意識到這些問題的敏感性，並提供了一份諒解備忘錄。

## 諒解備忘錄

Linux 內核社區深刻理解在不同作業系統供應商、發行商、硬體供應商和其他各方之間進行協調時，保持硬體安全問題處於限制狀態的要求。

Linux 內核社區在過去已經成功地處理了硬體安全問題，並且有必要的機制允許在限制限制下進行符合社區的開發。

Linux 內核社區有一個專門的硬體安全小組負責初始聯繫，並監督在限制規則下處理此類問題的過程。

硬體安全小組確定開發人員（領域專家），他們將組成特定問題的初始響應小組。最初的響應小組可以引入更多的開發人員（領域專家）以最佳的技術方式解決這個問題。

所有相關開發商承諾遵守限制規定，並對收到的信息保密。違反承諾將導致立即從當前問題中排除，並從所有相關郵件列表中刪除。此外，硬體安全小組還將把違反者排除在未來的問題之外。這一後果的影響在我們社區是一種非常有效的威懾。如果發生違規情況，硬體安全小組將立即通知相關方。如果您或任何人發現潛在的違規行為，請立即向硬體安全人員報告。

## 流程

由於 Linux 內核開發的全球分布式特性，面對面的會議幾乎不可能解決硬體安全問題。由於時區和其他因素，電話會議很難協調，只能在絕對必要時使用。加密電子郵件已被證明是解決此類問題的最有效和最安全的通信方法。

## 開始披露

披露內容首先通過電子郵件聯繫 Linux 內核硬體安全小組。此初始聯繫人應包含問題的描述和任何已知受影響硬體的列表。如果您的組織製造或分發受影響的硬體，我們建議您也考慮哪些其他硬體可能會受到影響。

硬體安全小組將提供一個特定於事件的加密郵件列表，用於與報告者進行初步討論、進一步披露和協調。

硬體安全小組將向披露方提供一份開發人員（領域專家）名單，在與開發人員確認他們將遵守本諒解備忘錄和文件化流程後，應首先告知開發人員有關該問題的信息。這些開發人員組成初始響應小組，並在初始接觸後負責處理問題。硬體安全小組支持響應小組，但不一定參與緩解開發過程。

雖然個別開發人員可能通過其僱主受到保密協議的保護，但他們不能以 Linux 內核開發人員的身份簽訂個別保密協議。但是，他們將同意遵守這一書面程序和諒解備忘錄。

披露方應提供已經或應該被告知該問題的所有其他實體的聯繫人名單。這有幾個目的：

- 披露的實體列表允許跨行業通信，例如其他作業系統供應商、硬體供應商等。
- 可聯繫已披露的實體，指定應參與緩解措施開發的專家。
- 如果需要處理某一問題的專家受僱於某一上市實體或某一上市實體的成員，則響應小組可要求該實體披露該專家。這確保專家也是實體反應小組的一部分。

### 披露

披露方通過特定的加密郵件列表向初始響應小組提供詳細信息。

根據我們的經驗，這些問題的技術文檔通常是一個足夠的起點，最好通過電子郵件進行進一步的技術澄清。

### 緩解開發

初始響應小組設置加密郵件列表，或在適當的情況下重新修改現有郵件列表。

使用郵件列表接近於正常的 Linux 開發過程，並且在過去已經成功地用於為各種硬體安全問題開發緩解措施。

郵件列表的操作方式與正常的 Linux 開發相同。發布、討論和審查修補程序，如果同意，則應用於非公共 git 存儲庫，參與開發人員只能通過安全連接訪問該存儲庫。存儲庫包含針對主線內核的主開發分支，並根據需要為穩定的內核版本提供向後移植分支。

最初的響應小組將根據需要從 Linux 內核開發人員社區中確定更多的專家。引進專家可以在開發過程中的任何時候發生，需要及時處理。

如果專家受僱於披露方提供的披露清單上的實體或其成員，則相關實體將要求其參與。

否則，披露方將被告知專家參與的情況。諒解備忘錄涵蓋了專家，要求披露方確認參與。如果披露方有令人信服的理由提出異議，則必須在五個工作日內提出異議，並立即與事件小組解決。如果披露方在五個工作日內未作出回應，則視為默許。

在確認或解決異議後，專家由事件小組披露，並進入開發過程。

### 協調發布

有關各方將協商限制結束的日期和時間。此時，準備好的緩解措施集成到相關的內核樹中並發布。

雖然我們理解硬體安全問題需要協調限制時間，但限制時間應限制在所有有關各方制定、測試和準備緩解措施所需的最短時間內。人為地延長限制時間以滿足會議討論日期或其他非技術原因，會給相關的開發人員和響應小組帶來了更多的工作和負擔，因為補丁需要保持最新，以便跟蹤正在進行的上游內核開發，這可能會造成衝突的更改。

### CVE 分配

硬體安全小組和初始響應小組都不分配 CVE，開發過程也不需要 CVE。如果 CVE 是由披露方提供的，則可用於文檔中。

## 流程專使

為了協助這一進程，我們在各組織設立了專使，他們可以回答有關報告流程和進一步處理的問題或提供指導。專使不參與特定問題的披露，除非響應小組或相關披露方提出要求。現任專使名單：

ARM	
AMD	Tom Lendacky < <a href="mailto:tom.lendacky@amd.com">tom.lendacky@amd.com</a> >
IBM	
Intel	Tony Luck < <a href="mailto:tony.luck@intel.com">tony.luck@intel.com</a> >
Qualcomm	Trilok Soni < <a href="mailto:tsoni@codeaurora.org">tsoni@codeaurora.org</a> >
Microsoft	Sasha Levin < <a href="mailto:sashal@kernel.org">sashal@kernel.org</a> >
VMware	
Xen	Andrew Cooper < <a href="mailto:andrew.cooper3@citrix.com">andrew.cooper3@citrix.com</a> >
Canonical	John Johansen < <a href="mailto:john.johansen@canonical.com">john.johansen@canonical.com</a> >
Debian	Ben Hutchings < <a href="mailto:ben@decentient.org.uk">ben@decentient.org.uk</a> >
Oracle	Konrad Rzeszutek Wilk < <a href="mailto:konrad.wilk@oracle.com">konrad.wilk@oracle.com</a> >
Red Hat	Josh Poimboeuf < <a href="mailto:jpoimboe@redhat.com">jpoimboe@redhat.com</a> >
SUSE	Jiri Kosina < <a href="mailto:jkosina@suse.cz">jkosina@suse.cz</a> >
Amazon	
Google	Kees Cook < <a href="mailto:keescook@chromium.org">keescook@chromium.org</a> >

如果要將您的組織添加到專使名單中，請與硬體安全小組聯繫。被提名的專使必須完全理解和支持我們的過程，並且在 Linux 內核社區中很容易聯繫。

## 加密郵件列表

我們使用加密郵件列表進行通信。這些列表的工作原理是，發送到列表的電子郵件使用列表的 PGP 密鑰或列表的/MIME 證書進行加密。郵件列表軟體對電子郵件進行解密，並使用訂閱者的 PGP 密鑰或 S/MIME 證書為每個訂閱者分別對其進行重新加密。有關郵件列表軟體和用於確保列表安全和數據保護的設置的詳細信息，請訪問: <https://www.kernel.org/>…

## 關鍵點

初次接觸見 [接觸](#)。對於特定於事件的郵件列表，密鑰和 S/MIME 證書通過特定列表發送的電子郵件傳遞給訂閱者。

### 訂閱事件特定列表

訂閱由響應小組處理。希望參與通信的披露方將潛在訂戶的列表發送給響應組，以便響應組可以驗證訂閱請求。

每個訂戶都需要通過電子郵件向響應小組發送訂閱請求。電子郵件必須使用訂閱伺服器的 PGP 密鑰或 S/MIME 證書簽名。如果使用 PGP 密鑰，則必須從公鑰伺服器獲得該密鑰，並且理想情況下該密鑰連接到 Linux 內核的 PGP 信任網。另請參見: <https://www.kernel.org/signature.html>.

響應小組驗證訂閱者，並將訂閱者添加到列表中。訂閱後，訂閱者將收到來自郵件列表的電子郵件，該郵件列表使用列表的 PGP 密鑰或列表的 S/MIME 證書簽名。訂閱者的電子郵件客戶端可以從簽名中提取 PGP 密鑰或 S/MIME 證書，以便訂閱者可以向列表發送加密電子郵件。

這些是一些總體性技術指南，由於不大好分類而放在這裡：

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

---

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

---

### Original Documentation/process/magic-number.rst

如果想評論或更新本文的內容，請直接發信到 LKML。如果你使用英文交流有困難的話，也可以向中文版維護者求助。如果本翻譯更新不及時或者翻譯存在問題，請聯繫中文版維護者：

中文版維護者： 賈威威 Jia Wei Wei <[harryxiyou@gmail.com](mailto:harryxiyou@gmail.com)>  
中文版翻譯者： 賈威威 Jia Wei Wei <[harryxiyou@gmail.com](mailto:harryxiyou@gmail.com)>  
中文版校譯者： 賈威威 Jia Wei Wei <[harryxiyou@gmail.com](mailto:harryxiyou@gmail.com)>  
胡皓文 Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

### Linux 魔術數

這個文件是有關當前使用的魔術值註冊表。當你給一個結構添加了一個魔術值，你也應該把這個魔術值添加到這個文件，因為我們最好把用於各種結構的魔術值統一起來。

使用魔術值來保護內核數據結構是一個非常好的主意。這就允許你在運行期檢查 (a) 一個結構是否已經被攻擊，或者 (b) 你已經給一個例行程序通過了一個錯誤的結構。後一種情況特別地有用—特別是當你通過一個空指針指向結構體的時候。tty 源碼，例如，經常通過特定驅動使用這種方法並且反覆地排列特定方面的結構。

使用魔術值的方法是在結構的開始處聲明的，如下：

```
struct tty_ldisc {
    int     magic;
    ...
};
```

當你以後給內核添加增強功能的時候，請遵守這條規則！這樣就會節省數不清的調試時間，特別是一些古怪的情況，例如，數組超出範圍並且重新寫了超出部分。遵守這個規則，這些情況可以被快速地，安全地避免。

**Theodore Ts' o** 31 Mar 94

給當前的 Linux 2.1.55 添加魔術表。

Michael Chastain <<mailto:mec@shout.net>> 22 Sep 1997

現在應該最新的 Linux 2.1.112. 因為在特性凍結期間，不能在 2.2.x 前改變任何東西。這些條目被數域所排序。

Krzysztof G.Baranowski <[mailto: kgb@knm.org.pl](mailto:kgb@knm.org.pl)> 29 Jul 1998

更新魔術表到 Linux 2.5.45。剛好越過特性凍結，但是有可能還會有一些新的魔術值在 2.6.x 之前融入到內核中。

Petr Baudis <<mailto:pasky@ucw.cz>> 03 Nov 2002

更新魔術表到 Linux 2.5.74。

Fabian Frederick <<mailto:ffrederick@users.sourceforge.net>> 09 Jul 2003

魔術數名	數字	結構	文件
PG_MAGIC	'P'	pg_{read,write}_hdr	include/linux/pg.h
CMAGIC	0x0111	user	include/linux/a.out.
MKISS_DRIVER_MAGIC	0x04bf	mkiss_channel	drivers/net/mkiss.h
HDLC_MAGIC	0x239e	n_hdlc	drivers/char/n_hdlc.
APM BIOS MAGIC	0x4101	apm_user	arch/x86/kernel/apm_
DB_MAGIC	0x4442	fc_info	drivers/net/iph5526_
DL_MAGIC	0x444d	fc_info	drivers/net/iph5526_
FASYNC_MAGIC	0x4601	fasync_struct	include/linux/fs.h
FF_MAGIC	0x4646	fc_info	drivers/net/iph5526_
PTY_MAGIC	0x5001		drivers/char/pty.c
PPP_MAGIC	0x5002	ppp	include/linux/if_ppp
SSTATE_MAGIC	0x5302	serial_state	include/linux/serial
SLIP_MAGIC	0x5302	slip	drivers/net/slip.h
STRIP_MAGIC	0x5303	strip	drivers/net/strip.c
SIXPACK_MAGIC	0x5304	sixpack	drivers/net/hamradio
AX25_MAGIC	0x5316	ax_disp	drivers/net/mkiss.h
TTY_MAGIC	0x5401	tty_struct	include/linux/tty.h

Table 2 – continued from previous page

魔術數名	數字	結構	文件
MGSL_MAGIC	0x5401	mgsl_info	drivers/char/syncl...
TTY_DRIVER_MAGIC	0x5402	tty_driver	include/linux/tty_d...
MGS LPC_MAGIC	0x5402	mgslpc_info	drivers/char/pcmcia/
USB_SERIAL_MAGIC	0x6702	usb_serial	drivers/usb/serial/u...
FULL_DUPLEX_MAGIC	0x6969		drivers/net/ethernet...
USB_BLUETOOTH_MAGIC	0x6d02	usb_bluetooth	drivers/usb/class/bl...
RFCOMM_TTY_MAGIC	0x6d02		net/bluetooth/rfcomm...
USB_SERIAL_PORT_MAGIC	0x7301	usb_serial_port	drivers/usb/serial/u...
CG_MAGIC	0x00090255	ufs_cylinder_group	include/linux/ufs_fs...
LSEMAGIC	0x05091998	lse	drivers/fc4/fc.c
GDTIOCTL_MAGIC	0x06030f07	gdth_iowr_str	drivers/scsi/gdth_i...
RIEBL_MAGIC	0x09051990		drivers/net/atarilan...
NBD_REQUEST_MAGIC	0x12560953	nbd_request	include/linux/nbd.h
RED_MAGIC2	0x170fc2a5	(any)	mm/slab.c
BAYCOM_MAGIC	0x19730510	baycom_state	drivers/net/baycom_e...
ISDN_X25IFACE_MAGIC	0x1e75a2b9	isdn_x25iface_proto_data	drivers/isdn/isdn_x2...
ECP_MAGIC	0x21504345	cdkecpsig	include/linux/cdk.h
LSOMAGIC	0x27091997	lso	drivers/fc4/fc.c
LSMAGIC	0x2a3b4d2a	ls	drivers/fc4/fc.c
WANPIPE_MAGIC	0x414C4453	sdla_{dump,exec}	include/linux/wanpip...
CS_CARD_MAGIC	0x43525553	cs_card	sound/oss/cs46xx.c
LABELCL_MAGIC	0x4857434c	labelcl_info_s	include/asm/ia64/sn/
ISDN_ASYNC_MAGIC	0x49344C01	modem_info	include/linux/isdn.h
CTC_ASYNC_MAGIC	0x49344C01	ctc_tty_info	drivers/s390/net/ctc...
ISDN_NET_MAGIC	0x49344C02	isdn_net_local_s	drivers/isdn/i4l/isdn...
SAVEKMSG_MAGIC2	0x4B4D5347	savekmsg	arch/*/amiga/config...
CS_STATE_MAGIC	0x4c4f4749	cs_state	sound/oss/cs46xx.c
SLAB_C_MAGIC	0x4f17a36d	kmem_cache	mm/slab.c
COW_MAGIC	0x4f4f4f4d	cow_header_v1	arch/um/drivers/ubd...
I810_CARD_MAGIC	0x5072696E	i810_card	sound/oss/i810_audio...
TRIDENT_CARD_MAGIC	0x5072696E	trident_card	sound/oss/trident.c
ROUTER_MAGIC	0x524d4157	wan_device	[in wanrouter.h pre 3...
SAVEKMSG_MAGIC1	0x53415645	savekmsg	arch/*/amiga/config...
GDA_MAGIC	0x58464552	gda	arch/mips/include/as...
RED_MAGIC1	0x5a2cf071	(any)	mm/slab.c
EEPROM_MAGIC_VALUE	0x5ab478d2	lanai_dev	drivers/atm/lanai.c
HDLCDRV_MAGIC	0x5ac6e778	hdlcdrv_state	include/linux/hlcdcr...
PCXX_MAGIC	0x5c6df104	channel	drivers/char/pcxx.h

con

Table 2 – continued from previous page

魔術數名	數字	結構	文件
KV_MAGIC	0x5f4b565f	kernel_vars_s	arch/mips/include/as
I810_STATE_MAGIC	0x63657373	i810_state	sound/oss/i810_audio
TRIDENT_STATE_MAGIC	0x63657373	trient_state	sound/oss/trident.c
M3_CARD_MAGIC	0x646e6f50	m3_card	sound/oss/maestro3.c
FW_HEADER_MAGIC	0x65726F66	fw_header	drivers/atm/fore200e
SLOT_MAGIC	0x67267321	slot	drivers/hotplug/cpqpc
SLOT_MAGIC	0x67267322	slot	drivers/hotplug/acpi
LO_MAGIC	0x68797548	nbd_device	include/linux/nbd.h
M3_STATE_MAGIC	0x734d724d	m3_state	sound/oss/maestro3.c
VMALLOC_MAGIC	0x87654320	snd_alloc_track	sound/core/memory.c
KMALLOC_MAGIC	0x87654321	snd_alloc_track	sound/core/memory.c
PWC_MAGIC	0x89DC10AB	pwc_device	drivers/usb/media/pwc
NBD_REPLY_MAGIC	0x96744668	nbd_reply	include/linux/nbd.h
ENI155_MAGIC	0xa54b872d	midway_eprom	drivers/atm/eni.h
CODA_MAGIC	0xC0DAC0DA	coda_file_info	fs/coda/coda_fs_i.h
DPMEM_MAGIC	0xc0ffee11	gdt_pci_sram	drivers/scsi/gdth.h
YAM_MAGIC	0xF10A7654	yam_port	drivers/net/hamradio
CCB_MAGIC	0xf2691ad2	ccb	drivers/scsi/ncr53c8
QUEUE_MAGIC_FREE	0xf7e1c9a3	queue_entry	drivers/scsi/arm/que
QUEUE_MAGIC_USED	0xf7e1cc33	queue_entry	drivers/scsi/arm/que
HTB_CMAGIC	0xFEFAFEF1	htb_class	net/sched/sch_htb.c
NMI_MAGIC	0x48414d4d455201	nmi_s	arch/mips/include/as

請注意，在聲音記憶管理中仍然有一些特殊的為每個驅動定義的魔術值。查看 include/sound/sndmagic.h 來獲取他們完整的列表信息。很多 OSS 聲音驅動擁有自己從音效卡 PCI ID 構建的魔術值-他們也沒有被列在這裡。

IrDA 子系統也使用了大量的自己的魔術值，查看 include/net/irda/irda.h 來獲取他們完整的信息。

HFS 是另外一個比較大的使用魔術值的文件系統-你可以在 fs/hfs/hfs.h 中找到他們。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

### Original Documentation/process/volatile-considered-harmful.rst

如果想評論或更新本文的內容，請直接聯繫原文檔的維護者。如果你使用英文交流有困難的話，也可以向中文版維護者求助。如果本翻譯更新不及時或者翻譯存在問題，請聯繫中文版維護者：

英文版維護者： Jonathan Corbet <corbet@lwn.net>  
中文版維護者： 伍鵬 Bryan Wu <bryan.wu@analog.com>  
中文版翻譯者： 伍鵬 Bryan Wu <bryan.wu@analog.com>  
中文版校譯者： 張漢輝 Eugene Teo <eugeneteo@kernel.sg>  
楊瑞 Dave Young <hidave.darkstar@gmail.com>  
時奎亮 Alex Shi <alex.shi@linux.alibaba.com>  
胡皓文 Hu Haowen <src.res@email.cn>

### 為什麼不應該使用「volatile」類型

C 程式設計師通常認為 volatile 表示某個變量可以在當前執行的線程之外被改變；因此，在內核中用到共享數據結構時，常常會有 C 程式設計師喜歡使用 volatile 這類變量。換句話說，他們經常會把 volatile 類型看成某種簡易的原子變量，當然它們不是。在內核中使用 volatile 幾乎總是錯誤的；本文檔將解釋為什麼這樣。

理解 volatile 的關鍵是知道它的目的是用來消除優化，實際上很少有人真正需要這樣的應用。在內核中，程式設計師必須防止意外的並發訪問破壞共享的數據結構，這其實是一個完全不同的任務。用來防止意外並發訪問的保護措施，可以更加高效的避免大多數優化相關的問題。

像 volatile 一樣，內核提供了很多原語來保證並發訪問時的數據安全（自旋鎖，互斥量，內存屏障等等），同樣可以防止意外的優化。如果可以正確使用這些內核原語，那麼就沒有必要再使用 volatile。如果仍然必須使用 volatile，那麼幾乎可以肯定在代碼的某處有一個 bug。在正確設計的內核代碼中，volatile 能帶來的僅僅是使事情變慢。

思考一下這段典型的內核代碼：

```
spin_lock(&the_lock);
do_something_on(&shared_data);
do_something_else_with(&shared_data);
spin_unlock(&the_lock);
```

如果所有的代碼都遵循加鎖規則，當持有 the\_lock 的時候，不可能意外的改變 shared\_data 的值。任何可能訪問該數據的其他代碼都會在這個鎖上等待。自旋鎖原語跟內存屏障一樣——它們顯式的用來書寫成這樣——意味著數據訪問不會跨越它們而被優化。所以本來編譯器認爲它知道在 shared\_data 裡面將有什麼，但是因爲 spin\_lock() 調用跟內存屏障一樣，會強制編譯器忘記它所知道的一切。那麼在訪問這些數據時不會有優化的問題。

如果 shared\_data 被聲名爲 volatile，鎖操作將仍然是必須的。就算我們知道沒有其他人正在使用它，編譯器也將被阻止優化對臨界區內 shared\_data 的訪問。在鎖有效的同時，shared\_data 不是 volatile 的。在處理共享數據的時候，適當的鎖操作可以不再需要 volatile ——並且是有潛在危害的。

volatile 的存儲類型最初是爲那些內存映射的 I/O 寄存器而定義。在內核里，寄存器訪問也應該被鎖保護，但是人們也不希望編譯器「優化」臨界區內的寄存器訪問。內核里 I/O 的內存訪問是通過訪問函數完成的；不

贊成通過指針對 I/O 內存的直接訪問，並且不是在所有體系架構上都能工作。那些訪問函數正是為了防止意外優化而寫的，因此，再說一次，volatile 類型不是必需的。

另一種引起用戶可能使用 volatile 的情況是當處理器正忙著等待一個變量的值。正確執行一個忙等待的方法是：

```
while (my_variable != what_i_want)
    cpu_relax();
```

cpu\_relax() 調用會降低 CPU 的能量消耗或者讓位於超線程雙處理器；它也作為內存屏障一樣出現，所以，再一次，volatile 不是必需的。當然，忙等待一開始就是一種反常規的做法。

在內核中，一些稀少的情況下 volatile 仍然是有意義的：

- 在一些體系架構的系統上，允許直接的 I/O 內存訪問，那麼前面提到的訪問函數可以使用 volatile。基本上，每一個訪問函數調用它自己都是一個小的臨界區域並且保證了按照程式設計師期望的那樣發生訪問操作。
- 某些會改變內存的內聯彙編代碼雖然沒有什麼其他明顯的附作用，但是有被 GCC 刪除的可能性。在彙編聲明中加上 volatile 關鍵字可以防止這種刪除操作。
- Jiffies 變量是一種特殊情況，雖然每次引用它的時候都可以有不同的值，但讀 jiffies 變量時不需要任何特殊的加鎖保護。所以 jiffies 變量可以使用 volatile，但是不贊成其他跟 jiffies 相同類型變量使用 volatile。Jiffies 被認為是一種「愚蠢的遺留物”（Linus 的話）因為解決這個問題比保持現狀要麻煩的多。
- 由於某些 I/O 設備可能會修改連續一致的內存，所以有時，指向連續一致內存的數據結構的指針需要正確的使用 volatile。網絡適配器使用的環狀緩存區正是這類情形的一個例子，其中適配器用改變指針來表示哪些描述符已經處理過了。

對於大多代碼，上述幾種可以使用 volatile 的情況都不適用。所以，使用 volatile 是一種 bug 並且需要對這樣的代碼額外仔細檢查。那些試圖使用 volatile 的開發人員需要退一步想想他們真正想實現的是什麼。

非常歡迎刪除 volatile 變量的補丁—只要證明這些補丁完整的考慮了並發問題。

## 注釋

[1] <https://lwn.net/Articles/233481/> [2] <https://lwn.net/Articles/233482/>

## 致謝

最初由 Randy Dunlap 推動並作初步研究由 Jonathan Corbet 撰寫參考 Satyam Sharma，Johannes Stezenbach，Jesper Juhl，Heikki Orsila，H. Peter Anvin，Philipp Hahn 和 Stefan Richter 的意見改善了本檔。

TODOLIST:

- dev-tools/index

- doc-guide/index
- kernel-hacking/index
- trace/index
- maintainer/index
- fault-injection/index
- livepatch/index
- rust/index

### \* **內核 API 文檔**

以下手冊從內核開發人員的角度詳細介紹了特定的內核子系統是如何工作的。這裡的大部分信息都是直接從內核原始碼獲取的，並根據需要添加補充材料（或者至少是在我們設法添加的時候——可能不是所有的都是有需要的）。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

**Original** [.../.../cpu-freq/index](#)

**Translator** Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)> Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

### \* **Linux CPUFreq - Linux(TM) 內核中的 CPU 頻率和電壓升降代碼**

Author: Dominik Brodowski <[linux@brodo.de](mailto:linux@brodo.de)>

時鐘升降允許你在運行中改變 CPU 的時鐘速度。這是一個很好的節省電池電量的方法，因為時鐘速度越低，CPU 消耗的電量越少。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

---

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:src.res@email.cn)。

---

**Original** [.../cpu-freq/core](#)

**Translator** Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)> Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

## CPUFreq 核心和 CPUFreq 通知器的通用說明

作者：

- Dominik Brodowski <[linux@brodo.de](mailto:linux@brodo.de)>
- David Kimdon <[dwhedon@debian.org](mailto:dwhedon@debian.org)>
- Rafael J. Wysocki <[rafael.j.wysocki@intel.com](mailto:rafael.j.wysocki@intel.com)>
- Viresh Kumar <[viresh.kumar@linaro.org](mailto:viresh.kumar@linaro.org)>

### 1. CPUFreq 核心和接口

cpufreq 核心代碼位於 drivers/cpufreq/cpufreq.c 中。這些 cpufreq 代碼為 CPUFreq 架構的驅動程序（那些操作硬體切換頻率的代碼）以及“通知器”提供了一個標準化的接口。這些是設備驅動程序或需要了解策略變化的其它內核部分（如 ACPI 熱量管理）或所有頻率更改（除計時代碼外），甚至需要強制確定速度限制的通知器（如 ARM 架構上的 LCD 驅動程序）。此外，內核“常數”loops\_per\_jiffy 會根據頻率變化而更新。

cpufreq 策略的引用計數由 cpufreq\_cpu\_get 和 cpufreq\_cpu\_put 來完成，以確保 cpufreq 驅動程序被正確地註冊到核心中，並且驅動程序在 cpufreq\_put\_cpu 被調用之前不會被卸載。這也保證了每個 CPU 核的 cpufreq 策略在使用期間不會被釋放。

### 2. CPUFreq 通知器

CPUFreq 通知器符合標準的內核通知器接口。關於通知器的細節請參閱 [linux/include/linux/notifier.h](#)。這裡有兩個不同的 CPUfreq 通知器 - 策略通知器和轉換通知器。

## 2.1 CPUFreq 策略通知器

當創建或移除策略時，這些都會被通知。

階段是在通知器的第二個參數中指定的。當第一次創建策略時，階段是 CPUFREQ\_CREATE\_POLICY，當策略被移除時，階段是 CPUFREQ\_REMOVE\_POLICY。

第三個參數 void \*pointer 指向一個結構體 cpufreq\_policy，其包括 min，max(新策略的下限和上限（單位為 kHz）) 這幾個值。

## 2.2 CPUFreq 轉換通知器

當 CPUfreq 驅動切換 CPU 核心頻率時，策略中的每個在線 CPU 都會收到兩次通知，這些變化沒有任何外部干擾。

第二個參數指定階段 - CPUFREQ\_PRECHANGE or CPUFREQ\_POSTCHANGE.

第三個參數是一個包含如下值的結構體 cpufreq\_freqs：

cpu	受影響 cpu 的編號
old	舊頻率
new	新頻率
flags	cpufreq 驅動的標誌

## 3. 含有 Operating Performance Point (OPP) 的 CPUFreq 表的生成

關於 OPP 的細節請參閱 Documentation/power/opp.rst

**dev\_pm\_opp\_init\_cpufreq\_table** - 這個功能提供了一個隨時可用的轉換程序，用來將 OPP 層關於可用頻率的內部信息翻譯成一種容易提供給 cpufreq 的格式。

**Warning:** 不要在中斷上下文中使用此函數。

例如：

```
soc_pm_init()
{
    /* Do things */
    r = dev_pm_opp_init_cpufreq_table(dev, &freq_table);
    if (!r)
        policy->freq_table = freq_table;
    /* Do other things */
}
```

---

**Note:** 該函數只有在 CONFIG\_PM\_OPP 之外還啓用了 CONFIG\_CPU\_FREQ 時才可用。

---

**dev\_pm\_opp\_free\_cpufreq\_table** 釋放 dev\_pm\_opp\_init\_cpufreq\_table 分配的表。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

---

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:src.res@email.cn)。

---

**Original** ../../cpu-freq/cpu-drivers

**Translator** Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)> Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

## 如何實現一個新的 CPUFreq 處理器驅動程序？

作者：

- Dominik Brodowski <[linux@brodo.de](mailto:linux@brodo.de)>
- Rafael J. Wysocki <[rafael.j.wysocki@intel.com](mailto:rafael.j.wysocki@intel.com)>
- Viresh Kumar <[viresh.kumar@linaro.org](mailto:viresh.kumar@linaro.org)>

### 1. 怎麼做？

如此，你剛剛得到了一個全新的 CPU/晶片組及其數據手冊，並希望為這個 CPU/晶片組添加 cpufreq 支持？很好，這裡有一些至關重要的提示：

#### 1.1 初始化

首先，在 `_initcall_level_7` (`module_init()`) 或更靠後的函數中檢查這個內核是否運行在正確的 CPU 和正確的晶片組上。如果是，則使用 `cpufreq_register_driver()` 向 CPUfreq 核心層註冊一個 `cpufreq_driver` 結構體。

結構體 `cpufreq_driver` 應該包含什麼成員？

`.name` - 驅動的名字。

`.init` - 一個指向 `per-policy` 初始化函數的指針。

.verify - 一個指向” verification” 函數的指針。

.setpolicy 或.fast\_switch 或.target 或.target\_index - 差異見下文。

並且可選擇

.flags - cpufreq 核的提示。

.driver\_data - cpufreq 驅動程序的特定數據。

.get\_intermediate 和 target\_intermediate - 用於在改變 CPU 頻率時切換到穩定的頻率。

.get - 返回 CPU 的當前頻率。

.bios\_limit - 返回 HW/BIOS 對 CPU 的最大頻率限制值。

.exit - 一個指向 per-policy 清理函數的指針，該函數在 cpu 热插拔過程的 CPU\_POST\_DEAD 階段被調用。

.suspend - 一個指向 per-policy 暫停函數的指針，該函數在關中斷且在該策略的調節器停止後被調用。

.resume - 一個指向 per-policy 恢復函數的指針，該函數在關中斷且在調節器再一次開始前被調用。

.ready - 一個指向 per-policy 準備函數的指針，該函數在策略完全初始化之後被調用。

.attr - 一個指向 NULL 結尾的” struct freq\_attr” 列表的指針，該函數允許導出值到 sysfs。

.boost\_enabled - 如果設置，則啓用提升 (boost) 頻率。

.set\_boost - 一個指向 per-policy 函數的指針，該函數用來開啓/關閉提升 (boost) 頻率功能。

## 1.2 Per-CPU 初始化

每當一個新的 CPU 被註冊到設備模型中，或者在 cpufreq 驅動註冊自己之後，如果此 CPU 的 cpufreq 策略不存在，則會調用 per-policy 的初始化函數 cpufreq\_driver.init。請注意，.init() 和.exit() 程序只對策略調用一次，而不是對策略管理的每個 CPU 調用一次。它需要一個 struct cpufreq\_policy \*policy 作為參數。現在該怎麼做呢？

如果有必要，請在你的 CPU 上激活 CPUfreq 功能支持。

然後，驅動程序必須填寫以下數值：

policy->cpuinfo.min_freq 和 policy->cpuinfo.max_freq	該 CPU 支持的最低和最高頻率 (kHz)
policy->cpuinfo.transition_latency	CPU 在兩個頻率之間切換所需的時間，以納秒為單位 (如適用，否則指定 CPUFREQ_ETERNAL)
policy->cur	該 CPU 當前的工作頻率 (如適用)
policy->min, policy->max, policy->policy_and, if necessary, policy->governor	必須包含該 cpu 的「默認策略」。稍後會用這些值調用 cpufreq_driver.verify_and either cpufreq_driver.setpolicy or cpufreq_driver.target/target_index
policy->cpus	用與這個 CPU 一起做 DVFS 的 (在線 + 離線) CPU(即與它共享時鐘/電壓軌) 的掩碼更新這個

對於設置其中的一些值 (cpuinfo.min[max]\_freq, policy->min[max])，頻率表助手可能會有幫助。關於它們的更多信息，請參見第 2 節。

### 1.3 驗證

當用戶決定設置一個新的策略 (由「policy,governor,min,max 組成」) 時，必須對這個策略進行驗證，以便糾正不兼容的值。為了驗證這些值，cpufreq\_verify\_within\_limits(struct cpufreq\_policy \*policy, unsigned int min\_freq, unsigned int max\_freq) 函數可能會有幫助。關於頻率表助手的詳細內容請參見第 2 節。

您需要確保至少有一個有效頻率 (或工作範圍) 在 policy->min 和 policy->max 範圍內。如果有必要，先增加 policy->max，只有在沒有辦法的情況下，才減少 policy->min。

### 1.4 target 或 target\_index 或 setpolicy 或 fast\_switch?

大多數 cpufreq 驅動甚至大多數 cpu 頻率升降算法只允許將 CPU 頻率設置為預定義的固定值。對於這些，你可以使用->target()，->target\_index() 或->fast\_switch() 回調。

有些 cpufreq 功能的處理器可以自己在某些限制之間切換頻率。這些應使用->setpolicy() 回調。

### 1.5. target/target\_index

target\_index 調用有兩個參數：struct cpufreq\_policy \* policy`` 和 ``unsigned int 索引 (於列出的頻率表)。

當調用這裡時，CPUfreq 驅動必須設置新的頻率。實際頻率必須由 freq\_table[index].frequency 決定。它應該總是在錯誤的情況下恢復到之前的頻率 (即 policy->restore\_freq)，即使我們之前切換到中間頻率。

### 已棄用

目標調用有三個參數。`struct cpufreq_policy * policy, unsigned int target_frequency, unsigned int relation.`

CPUfreq 驅動在調用這裡時必須設置新的頻率。實際的頻率必須使用以下規則來確定。

- 繫跟“目標頻率”。
- `policy->min <= new_freq <= policy->max` (這必須是有效的!!!)
- 如果 `relation==CPUFREQ_REL_L`，嘗試選擇一個高於或等於 `target_freq` 的 `new_freq`。(“L 代表最低，但不能低於”)
- 如果 `relation==CPUFREQ_REL_H`，嘗試選擇一個低於或等於 `target_freq` 的 `new_freq`。(“H 代表最高，但不能高於”)

這裡，頻率表助手可能會幫助你-詳見第 2 節。

### 1.6. fast\_switch

這個函數用於從調度器的上下文進行頻率切換。並非所有的驅動都要實現它，因為不允許在這個回調中睡眠。這個回調必須經過高度優化，以儘可能快地進行切換。

這個函數有兩個參數：`struct cpufreq_policy *policy` 和 `unsigned int target_frequency`。

### 1.7 setpolicy

`setpolicy` 調用只需要一個```struct cpufreq_policy * policy```作為參數。需要將處理器內或晶片組內動態頻率切換的下限設置為 `policy->min`，上限設置為 `policy->max`，如果支持的話，當 `policy->policy` 為 `CPUFREQ_POLICY_PERFORMANCE` 時選擇面向性能的設置，當 `CPUFREQ_POLICY_POWERSAVE` 時選擇面向省電的設置。也可以查看 `drivers/cpufreq/longrun.c` 中的參考實現。

### 1.8 get\_intermediate 和 target\_intermediate

僅適用於 `target_index()` 和 `CPUFREQ_ASYNC_NOTIFICATION` 未設置的驅動。

`get_intermediate` 應該返回一個平台想要切換到的穩定的中間頻率，`target_intermediate()` 應該將 CPU 設置為該頻率，然後再跳轉到' index' 對應的頻率。核心會負責發送通知，驅動不必在 `target_intermediate()` 或 `target_index()` 中處理。

在驅動程序不想因為某個目標頻率切換到中間頻率的情況下，它們可以從 `get_intermediate()` 中返回' 0'。在這種情況下，核心將直接調用`->target_index()`。

注意：`->target_index()` 應該在失敗的情況下恢復到 `policy->restore_freq`，因為 core 會為此發送通知。

## 2. 頻率表助手

由於大多數 cpufreq 處理器只允許被設置為幾個特定的頻率，因此，一個帶有一些函數的「頻率表」可能會輔助處理器驅動程序的一些工作。這樣的“頻率表”由一個 cpufreq\_frequency\_table 條目構成的數組組成，”driver\_data”中包含了驅動程序的具體數值，”frequency”中包含了相應的頻率，並設置了標誌。在表的最後，需要添加一個 cpufreq\_frequency\_table 條目，頻率設置為 CPUFREQ\_TABLE\_END。而如果想跳過表中的一個條目，則將頻率設置為 CPUFREQ\_ENTRY\_INVALID。這些條目不需要按照任何特定的順序排序，但如果它們是 cpufreq 核心會對它們進行快速的 DVFS，因為搜索最佳匹配會更快。

如果策略在其 policy->freq\_table 欄位中包含一個有效的指針，cpufreq 表就會被核心自動驗證。

cpufreq\_frequency\_table\_verify() 保證至少有一個有效的頻率在 policy->min 和 policy->max 範圍內，並且所有其他標準都被滿足。這對->verify 調用很有幫助。

cpufreq\_frequency\_table\_target() 是對應於->target 階段的頻率表助手。只要把數值傳遞給這個函數，這個函數就會返回包含 CPU 要設置的頻率的頻率表條目。

以下宏可以作為 cpufreq\_frequency\_table 的疊代器。

cpufreq\_for\_each\_entry(pos, table) - 遍歷頻率表的所有條目。

cpufreq\_for\_each\_valid\_entry(pos, table) - 該函數遍歷所有條目，不包括 CPUFREQ\_ENTRY\_INVALID 頻率。使用參數 “pos” - 一個``cpufreq\_frequency\_table \* `` 作為循環變量，使用參數 “table” - 作為你想疊代的``cpufreq\_frequency\_table \* ``。

例如：

```
struct cpufreq_frequency_table *pos, *driver_freq_table;

cpufreq_for_each_entry(pos, driver_freq_table) {
    /* Do something with pos */
    pos->frequency = ...
}
```

如果你需要在 driver\_freq\_table 中處理 pos 的位置，不要減去指針，因為它的代價相當高。相反，使用宏 cpufreq\_for\_each\_entry\_idx() 和 cpufreq\_for\_each\_valid\_entry\_idx()。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

**Original** ../../cpu-freq/cpufreq-stats

**Translator** Yanteng Si <[siyanteng@loongson.cn](mailto:siyanteng@loongson.cn)> Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

## sysfs CPUFreq Stats 的一般說明

用戶信息

作者: Venkatesh Pallipadi <[venkatesh.pallipadi@intel.com](mailto:venkatesh.pallipadi@intel.com)>

### 1. 簡介

cpufreq-stats 是一個為每個 CPU 提供 CPU 頻率統計的驅動。這些統計數據在/sysfs 中以一堆只讀接口的形式提供。這個接口（在配置好後）將出現在 /sysfs (<sysfs root>/devices/system/cpu/cpuX/cpufreq/stats/) 中 cpufreq 下的一個單獨的目錄中，提供給每個 CPU。各種統計數據將在此目錄下形成只讀文件。

此驅動是獨立於任何可能運行在你所用 CPU 上的特定 cpufreq\_driver 而設計的。因此，它將與所有 cpufreq\_driver 一起工作。

### 2. 提供的統計數據 (舉例說明)

cpufreq stats 提供了以下統計數據（在下面詳細解釋）。

- time\_in\_state
- total\_trans
- trans\_table

所有的統計數據將從統計驅動被載入的時間（或統計被重置的時間）開始，到某一統計數據被讀取的時間為止。顯然，統計驅動不會有任何關於統計驅動載入之前的頻率轉換信息。

```
<mysystem>:/sys/devices/system/cpu/cpu0/cpufreq/stats # ls -l
total 0
drwxr-xr-x 2 root root 0 May 14 16:06 .
drwxr-xr-x 3 root root 0 May 14 15:58 ..
--w----- 1 root root 4096 May 14 16:06 reset
-r--r--r-- 1 root root 4096 May 14 16:06 time_in_state
-r--r--r-- 1 root root 4096 May 14 16:06 total_trans
-r--r--r-- 1 root root 4096 May 14 16:06 trans_table
```

- **reset**

只寫屬性，可用於重置統計計數器。這對於評估不同調節器下的系統行為非常有用，且無需重啓。

- **time\_in\_state**

此項給出了這個 CPU 所支持的每個頻率所花費的時間。cat 輸出的每一行都會有”<frequency> <time>”對，表示這個 CPU 在 <frequency> 上花費了 <time> 個 usertime 單位的時間。這裡的 usertime 單位是 10mS（類似於/proc 中輸出的其他時間）。

```
<mysystem>:/sys/devices/system/cpu/cpu0/cpufreq/stats # cat time_in_state
3600000 2089
3400000 136
3200000 34
3000000 67
2800000 172488
```

- **total\_trans**

給出了這個 CPU 上頻率轉換的總次數。cat 的輸出將有一個單一的計數，這就是頻率轉換的總數。

```
<mysystem>:/sys/devices/system/cpu/cpu0/cpufreq/stats # cat total_trans
20
```

- **trans\_table**

這將提供所有 CPU 頻率轉換的細粒度信息。這裡的 cat 輸出是一個二維矩陣，其中一個條目  $\langle i, j \rangle$  (第  $i$  行，第  $j$  列) 代表從  $\text{Freq}_i$  到  $\text{Freq}_j$  的轉換次數。 $\text{Freq}_i$  行和  $\text{Freq}_j$  列遵循驅動最初提供給 cpufreq 核的頻率表的排序順序，因此可以排序 (升序或降序) 或不排序。這裡的輸出也包含了每行每列的實際頻率值，以便更好地閱讀。

如果轉換表大於 PAGE\_SIZE，讀取時將返回一個-EFBIG 錯誤。

```
<mysystem>:/sys/devices/system/cpu/cpu0/cpufreq/stats # cat trans_table
From : To
      : 3600000 3400000 3200000 3000000 2800000
3600000:    0      5      0      0      0
3400000:    4      0      2      0      0
3200000:    0      1      0      2      0
3000000:    0      0      1      0      3
2800000:    0      0      0      2      0
```

### 3. 配置 cpufreq-stats

要在你的內核中配置 cpufreq-stats:

```
Config Main Menu
  Power management options (ACPI, APM) --->
    CPU Frequency scaling --->
      [*] CPU Frequency scaling
      [*] CPU frequency translation statistics
```

“CPU Frequency scaling” (CONFIG\_CPU\_FREQ) 應該被啓用以配置 cpufreq-stats。

“CPU frequency translation statistics” (CONFIG\_CPU\_FREQ\_STAT) 提供了包括 time\_in\_state、total\_trans 和 trans\_table 的統計數據。

一旦啓用了這個選項，並且你的 CPU 支持 cpufreq，你就可以在/sysfs 中看到 CPU 頻率統計。

### 郵件列表

這裡有一個 CPU 頻率變化的 CVS 提交和通用列表，您可以在這裡報告 bug、問題或提交補丁。要發布消息，請發送電子郵件到 [linux-pm@vger.kernel.org](mailto:linux-pm@vger.kernel.org)。

### 連結

FTP 檔案: \* <ftp://ftp.linux.org.uk/pub/linux/cpufreq/>

如何訪問 CVS 倉庫: \* <http://cvs.arm.linux.org.uk/>

CPUFreq 郵件列表: \* <http://vger.kernel.org/vger-lists.html#linux-pm>

SA-1100 的時鐘和電壓標度: \* <http://www.lartmaker.nl/projects/scaling>

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

**Original** Documentation/filesystems/index.rst

**Translator** Wang      Wenhua      <[wenhu.wang@vivo.com](mailto:wenhu.wang@vivo.com)>      Hu      Haowen  
<[src.res@email.cn](mailto:<src.res@email.cn>)>

### \* Linux Kernel 中的文件系統

這份正在開發的手冊或許在未來某個輝煌的日子裡以易懂的形式將 Linux 虛擬文件系統（VFS）層以及基於其上的各種文件系統如何工作呈現給大家。當前可以看到下面的內容。

### 文件系統

文件系統實現文檔。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:src.res@email.cn)。

## Original Documentation/filesystems/virtiofs.rst

譯者

中文版維護者： 王文虎 Wang Wenh [<wenhu.wang@vivo.com>](mailto:wenhu.wang@vivo.com)

中文版翻譯者： 王文虎 Wang Wenh [<wenhu.wang@vivo.com>](mailto:wenhu.wang@vivo.com)

中文版校譯者： 王文虎 Wang Wenh [<wenhu.wang@vivo.com>](mailto:wenhu.wang@vivo.com)

中文版校譯者： 王文虎 Wang Wenh [<wenhu.wang@vivo.com>](mailto:wenhu.wang@vivo.com)

繁體中文版校譯者：胡皓文 Hu Haowen [<src.res@email.cn>](mailto:src.res@email.cn)

## virtiofs: virtio-fs 主機 <-> 客機共享文件系統

- Copyright (C) 2020 Vivo Communication Technology Co. Ltd.

### 介紹

Linux 的 virtiofs 文件系統實現了一個半虛擬化 VIRTIO 類型「virtio-fs」設備的驅動，通過該類型設備實現客機 <-> 主機文件系統共享。它允許客機掛載一個已經導出到主機的目錄。

客機通常需要訪問主機或者遠程系統上的文件。使用場景包括：在新客機安裝時讓文件對其可見；從主機上的根文件系統啓動；對無狀態或臨時客機提供持久存儲和在客機之間共享目錄。

儘管在某些任務可能通過使用已有的網絡文件系統完成，但是卻需要非常難以自動化的配置步驟，且將存儲網絡暴露給客機。而 virtio-fs 設備通過提供不經過網絡的文件系統訪問文件的設計方式解決了這些問題。

另外，virtio-fs 設備發揮了主客機共存的優點提高了性能，並且提供了網絡文件系統所不具備的一些語義功能。

### 用法

以``myfs``標籤將文件系統掛載到``/mnt``：

```
guest# mount -t virtiofs myfs /mnt
```

請查閱 <https://virtio-fs.gitlab.io/> 了解配置 QEMU 和 virtiofsd 守護程序的詳細信息。

### 內幕

由於 virtio-fs 設備將 FUSE 協議用於文件系統請求，因此 Linux 的 virtiofs 文件系統與 FUSE 文件系統客戶端緊密集成在一起。客機充當 FUSE 客戶端而主機充當 FUSE 伺服器，內核與用戶空間之間的/dev/fuse 接口由 virtio-fs 設備接口代替。

FUSE 請求被置於虛擬隊列中由主機處理。主機填充緩衝區中的響應部分，而客機處理請求的完成部分。

將/dev/fuse 映射到虛擬隊列需要解決/dev/fuse 和虛擬隊列之間語義上的差異。每次讀取/dev/fuse 設備時，FUSE 客戶端都可以選擇要傳輸的請求，從而可以使某些請求優先於其他請求。虛擬隊列有其隊列語義，無法更改已入隊請求的順序。在虛擬隊列已滿的情況下尤其關鍵，因為此時不可能加入高優先級的請求。為了解決此差異，virtio-fs 設備採用「hiprio」（高優先級）虛擬隊列，專門用於有別於普通請求的高優先級請求。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

**Original** [.../..../filesystems/debugfs](#)

### Debugfs

譯者

中文版維護者：羅楚成 Chucheng Luo <[luochucheng@vivo.com](mailto:luochucheng@vivo.com)>

中文版翻譯者：羅楚成 Chucheng Luo <[luochucheng@vivo.com](mailto:luochucheng@vivo.com)>

中文版校譯者：羅楚成 Chucheng Luo <[luochucheng@vivo.com](mailto:luochucheng@vivo.com)>

繁體中文版校譯者：胡皓文 Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

版權所有 2020 羅楚成 <[luochucheng@vivo.com](mailto:luochucheng@vivo.com)> 版權所有 2021 胡皓文 Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

Debugfs 是內核開發人員在用戶空間獲取信息的簡單方法。與/proc 不同，proc 只提供進程信息。也不像 sysfs，具有嚴格的「每個文件一個值」的規則。debugfs 根本沒有規則，開發人員可以在這裡放置他們想要的任何信息。debugfs 文件系統也不能用作穩定的 ABI 接口。從理論上講，debugfs 導出文件的時候沒有任何約束。但是 [1] 實際情況並不總是那麼簡單。即使是 debugfs 接口，也最好根據需要進行設計，並儘量保持接口不變。

Debugfs 通常使用以下命令安裝：

```
mount -t debugfs none /sys/kernel/debug
```

(或等效的/etc/fstab 行)。debugfs 根目錄默認僅可由 root 用戶訪問。要更改對文件樹的訪問，請使用「uid」，「gid」和「mode」掛載選項。請注意，debugfs API 僅按照 GPL 協議導出到模塊。

使用 debugfs 的代碼應包含 <linux/debugfs.h>。然後，首先是創建至少一個目錄來保存一組 debugfs 文件：

```
struct dentry *debugfs_create_dir(const char *name, struct dentry *parent);
```

如果成功，此調用將在指定的父目錄下創建一個名為 name 的目錄。如果 parent 參數為空，則會在 debugfs 根目錄中創建。創建目錄成功時，返回值是一個指向 dentry 結構體的指針。該 dentry 結構體的指針可用於在目錄中創建文件（以及最後將其清理乾淨）。ERR\_PTR (-ERROR) 返回值表明出錯。如果返回 ERR\_PTR (-ENODEV)，則表明內核是在沒有 debugfs 支持的情況下構建的，並且下述函數都不會起作用。

在 debugfs 目錄中創建文件的最通用方法是：

```
struct dentry *debugfs_create_file(const char *name, umode_t mode,
                                   struct dentry *parent, void *data,
                                   const struct file_operations *fops);
```

在這裡，name 是要創建的文件的名稱，mode 描述了訪問文件應具有的權限，parent 指向應該保存文件的目錄，data 將存儲在產生的 inode 結構體的 i\_private 欄位中，而 fops 是一組文件操作函數，這些函數中實現文件操作的具體行為。至少，read () 和/或 write () 操作應提供；其他可以根據需要包括在內。同樣的，返回值將是指向創建文件的 dentry 指針，錯誤時返回 ERR\_PTR (-ERROR)，系統不支持 debugfs 時返回值為 ERR\_PTR (-ENODEV)。創建一個初始大小的文件，可以使用以下函數代替：

```
struct dentry *debugfs_create_file_size(const char *name, umode_t mode,
                                       struct dentry *parent, void *data,
                                       const struct file_operations *fops,
                                       loff_t file_size);
```

file\_size 是初始文件大小。其他參數跟函數 debugfs\_create\_file 的相同。

在許多情況下，沒必要自己去創建一組文件操作；對於一些簡單的情況，debugfs 代碼提供了許多幫助函數。包含單個整數值的文件可以使用以下任何一項創建：

```
void debugfs_create_u8(const char *name, umode_t mode,
                      struct dentry *parent, u8 *value);
void debugfs_create_u16(const char *name, umode_t mode,
                      struct dentry *parent, u16 *value);
struct dentry *debugfs_create_u32(const char *name, umode_t mode,
                                 struct dentry *parent, u32 *value);
void debugfs_create_u64(const char *name, umode_t mode,
                      struct dentry *parent, u64 *value);
```

這些文件支持讀取和寫入給定值。如果某個文件不支持寫入，只需根據需要設置 mode 參數位。這些文件中

的值以十進位表示；如果需要使用十六進位，可以使用以下函數替代：

```
void debugfs_create_x8(const char *name, umode_t mode,
                      struct dentry *parent, u8 *value);
void debugfs_create_x16(const char *name, umode_t mode,
                      struct dentry *parent, u16 *value);
void debugfs_create_x32(const char *name, umode_t mode,
                      struct dentry *parent, u32 *value);
void debugfs_create_x64(const char *name, umode_t mode,
                      struct dentry *parent, u64 *value);
```

這些功能只有在開發人員知道導出值的大小的時候才有用。某些數據類型在不同的架構上有不同的寬度，這樣會使情況變得有些複雜。在這種特殊情況下可以使用以下函數：

```
void debugfs_create_size_t(const char *name, umode_t mode,
                           struct dentry *parent, size_t *value);
```

不出所料，此函數將創建一個 debugfs 文件來表示類型為 size\_t 的變量。

同樣地，也有導出無符號長整型變量的函數，分別以十進位和十六進位表示如下：

```
struct dentry *debugfs_create_ulong(const char *name, umode_t mode,
                                    struct dentry *parent,
                                    unsigned long *value);
void debugfs_create_xul(const char *name, umode_t mode,
                      struct dentry *parent, unsigned long *value);
```

布爾值可以通過以下方式放置在 debugfs 中：

```
struct dentry *debugfs_create_bool(const char *name, umode_t mode,
                                   struct dentry *parent, bool *value);
```

讀取結果文件將產生 Y（對於非零值）或 N，後跟換行符寫入的時候，它只接受大寫或小寫值或 1 或 0。任何其他輸入將被忽略。

同樣，atomic\_t 類型的值也可以放置在 debugfs 中：

```
void debugfs_create_atomic_t(const char *name, umode_t mode,
                            struct dentry *parent, atomic_t *value)
```

讀取此文件將獲得 atomic\_t 值，寫入此文件將設置 atomic\_t 值。

另一個選擇是通過以下結構體和函數導出一個任意二進位數據塊：

```
struct debugfs_blob_wrapper {
    void *data;
    unsigned long size;
};

struct dentry *debugfs_create_blob(const char *name, umode_t mode,
```

```
    struct dentry *parent,
    struct debugfs_blob_wrapper *blob);
```

讀取此文件將返回由指針指向 `debugfs_blob_wrapper` 結構體的數據。一些驅動使用「blobs」作為一種返回幾行（靜態）格式化文本的簡單方法。這個函數可用於導出二進位信息，但似乎在主線中沒有任何代碼這樣做。請注意，使用 `debugfs_create_blob()` 命令創建的所有文件是只讀的。

如果您要轉儲一個寄存器塊（在開發過程中經常會這麼做，但是這樣的調試代碼很少上傳到主線中。Debugfs 提供兩個函數：一個用於創建僅寄存器文件，另一個把一個寄存器塊插入一個順序文件中：

```
struct debugfs_reg32 {
    char *name;
    unsigned long offset;
};

struct debugfs_regset32 {
    struct debugfs_reg32 *regs;
    int nregs;
    void __iomem *base;
};

struct dentry *debugfs_create_regset32(const char *name, umode_t mode,
                                       struct dentry *parent,
                                       struct debugfs_regset32 *regset);

void debugfs_print_regs32(struct seq_file *s, struct debugfs_reg32 *regs,
                           int nregs, void __iomem *base, char *prefix);
```

「`base`」參數可能為 0，但您可能需要使用 `_stringify` 構建 `reg32` 數組，實際上有許多寄存器名稱（宏）是寄存器塊在基址上的字節偏移量。

如果要在 `debugfs` 中轉儲 `u32` 數組，可以使用以下函數創建文件：

```
void debugfs_create_u32_array(const char *name, umode_t mode,
                             struct dentry *parent,
                             u32 *array, u32 elements);
```

「`array`」參數提供數據，而「`elements`」參數為數組中元素的數量。注意：數組創建後，數組大小無法更改。

有一個函數來創建與設備相關的 `seq_file`：

```
struct dentry *debugfs_create_devm_seqfile(struct device *dev,
                                           const char *name,
                                           struct dentry *parent,
                                           int (*read_fn)(struct seq_file *s,
                                                          void *data));
```

「`dev`」參數是與此 `debugfs` 文件相關的設備，並且「`read_fn`」是一個函數指針，這個函數在列印 `seq_file` 內容的時候被回調。

還有一些其他的面向目錄的函數：

```
struct dentry *debugfs_rename(struct dentry *old_dir,
                           struct dentry *old_dentry,
                           struct dentry *new_dir,
                           const char *new_name);

struct dentry *debugfs_create_symlink(const char *name,
                           struct dentry *parent,
                           const char *target);
```

調用 `debugfs_rename()` 將為現有的 `debugfs` 文件重命名，可能同時切換目錄。`new_name` 函數調用之前不能存在；返回值為 `old_dentry`，其中包含更新的信息。可以使用 `debugfs_create_symlink()` 創建符號連結。

所有 `debugfs` 用戶必須考慮的一件事是：

`debugfs` 不會自動清除在其中創建的任何目錄。如果一個模塊在不顯式刪除 `debugfs` 目錄的情況下卸載模塊，結果將會遺留很多野指針，從而導致系統不穩定。因此，所有 `debugfs` 用戶-至少是那些可以作為模塊構建的用戶-必須做模塊卸載的時候準備刪除在此創建的所有文件和目錄。一份文件可以通過以下方式刪除：

```
void debugfs_remove(struct dentry *dentry);
```

`dentry` 值可以為 `NULL` 或錯誤值，在這種情況下，不會有任何文件被刪除。

很久以前，內核開發者使用 `debugfs` 時需要記錄他們創建的每個 `dentry` 指針，以便最後所有文件都可以被清理掉。但是，現在 `debugfs` 用戶能調用以下函數遞歸清除之前創建的文件：

```
void debugfs_remove_recursive(struct dentry *dentry);
```

如果將對應頂層目錄的 `dentry` 傳遞給以上函數，則該目錄下的整個層次結構將會被刪除。

注釋：[1] <http://lwn.net/Articles/309298/>

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：`<src.res@email.cn>`。

**Original** Documentation/filesystems/tmpfs.rst

Translated by Wang Qing <[wangqing@vivo.com](mailto:wangqing@vivo.com)> and Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

## Tmpfs

Tmpfs 是一個將所有文件都保存在虛擬內存中的文件系統。

tmpfs 中的所有內容都是臨時的，也就是說沒有任何文件會在硬碟上創建。如果卸載 tmpfs 實例，所有保存在其中的文件都會丟失。

tmpfs 將所有文件保存在內核緩存中，隨著文件內容增長或縮小可以將不需要的頁面 swap 出去。它具有最大限制，可以通過「mount -o remount …」調整。

和 ramfs (創建 tmpfs 的模板) 相比，tmpfs 包含交換和限制檢查。和 tmpfs 相似的另一個東西是 RAM 磁碟 (/dev/ram\*)，可以在物理 RAM 中模擬固定大小的硬碟，並在此之上創建一個普通的文件系統。Ramdisks 無法 swap，因此無法調整它們的大小。

由於 tmpfs 完全保存於頁面緩存和 swap 中，因此所有 tmpfs 頁面將在/proc/meminfo 中顯示為「Shmem」，而在 free(1) 中顯示為「Shared」。請注意，這些計數還包括共享內存 (shmem，請參閱 ipcs(1))。獲得計數的最可靠方法是使用 df(1) 和 du(1)。

tmpfs 具有以下用途：

- 1) 內核總有一個無法看到的內部掛載，用於共享匿名映射和 SYSV 共享內存。

掛載不依賴於 CONFIG\_TMPFS。如果 CONFIG\_TMPFS 未設置，tmpfs 對用戶不可見。但是內部機制始終存在。

- 2) glibc 2.2 及更高版本期望將 tmpfs 掛載在/dev/shm 上以用於 POSIX 共享內存 (shm\_open, shm\_unlink)。添加內容到/etc/fstab 應注意如下：

```
tmpfs /dev/shm tmpfs defaults 0 0
```

使用時需要記住創建掛載 tmpfs 的目錄。

SYSV 共享內存無需掛載，內部已默認支持。(在 2.3 內核版本中，必須掛載 tmpfs 的前身 (shm fs) 才能使用 SYSV 共享內存)

- 3) 很多人（包括我）都覺的在/tmp 和/var/tmp 上掛載非常方便，並具有較大的 swap 分區。目前循環掛載 tmpfs 可以正常工作，所以大多數發布都應當可以使用 mkinitrd 通過/tmp 訪問/tmp。

- 4) 也許還有更多我不知道的地方:-)

tmpfs 有三個用於調整大小的掛載選項：

size	tmpfs 實例分配的字節數限制。默認值是不 swap 時物理 RAM 的一半。如果 tmpfs 實例過大，機器將死鎖，因為 OOM 處理將無法釋放該內存。
nr_blocks	與 size 相同，但以 PAGE_SIZE 為單位。
nr_inodes	tmpfs 實例的最大 inode 個數。默認值是物理內存頁數的一半，或者 (有高端內存的機器) 低端內存 RAM 的頁數，二者以較低者為準。

這些參數接受後綴 k, m 或 g 表示千，兆和千兆字節，可以在 remount 時更改。size 參數也接受後綴%用來限制 tmpfs 實例占用物理 RAM 的百分比：未指定 size 或 nr\_blocks 時，默認值為 size=50%

如果 `nr_blocks=0` (或 `size=0`)，`block` 個數將不受限制；如果 `nr_inodes=0`，`inode` 個數將不受限制。這樣掛載通常是不明智的，因為它允許任何具有寫權限的用戶通過訪問 `tmpfs` 耗盡機器上的所有內存；但同時這樣做也會增強在多個 CPU 的場景下的訪問。

`tmpfs` 具有為所有文件設置 NUMA 內存分配策略掛載選項 (如果啓用了 `CONFIG_NUMA`)，可以通過「`mount -o remount ...`」調整

<code>mpol=default</code>	採用進程分配策略 (請參閱 <code>set_mempolicy(2)</code> )
<code>mpol=prefer:Node</code>	傾向從給定的節點分配
<code>mpol=bind:NodeList</code>	只允許從指定的鍊表分配
<code>mpol=interleave</code>	傾向於依次從每個節點分配
<code>mpol=interleave:NodeList</code>	依次從每個節點分配
<code>mpol=local</code>	優先本地節點分配內存

`NodeList` 格式是以逗號分隔的十進位數字表示大小和範圍，最大和最小範圍是用- 分隔符的十進位數來表示。例如，`mpol=bind0-3,5,7,9-15`

帶有有效 `NodeList` 的內存策略將按指定格式保存，在創建文件時使用。當任務在該文件系統上創建文件時，會使用到掛載時的內存策略 `NodeList` 選項，如果設置的話，由調用任務的 `cpuset`[請參見 `Documentation/admin-guide/cgroup-v1/cpusets.rst`] 以及下面列出的可選標誌約束。如果 `NodeLists` 為設置為空集，則文件的內存策略將恢復為「默認」策略。

NUMA 內存分配策略有可選標誌，可以用於模式結合。在掛載 `tmpfs` 時指定這些可選標誌可以在 `NodeList` 之前生效。`Documentation/admin-guide/mm/numa_memory_policy.rst` 列出所有可用的內存分配策略模式標誌及其對內存策略。

<code>=static</code>	相當於	<code>MPOL_F_STATIC_NODES</code>
<code>=relative</code>	相當於	<code>MPOL_F_RELATIVE_NODES</code>

例如，`mpol=bind=staticNodeList` 相當於 `MPOL_BIND|MPOL_F_STATIC_NODES` 的分配策略

請注意，如果內核不支持 NUMA，那麼使用 `mpol` 選項掛載 `tmpfs` 將會失敗；`nodeList` 指定不在線的節點也會失敗。如果您的系統依賴於此，但內核會運行不帶 NUMA 功能 (也許是安全 `revocery` 內核)，或者具有較少的節點在線，建議從自動模式中省略 `mpol` 選項掛載選項。可以在以後通過「`mount -o remount,mpol=Policy:NodeList MountPoint`」添加到掛載點。

要指定初始根目錄，可以使用如下掛載選項：

模式	權限用八進位數字表示
<code>uid</code>	用戶 ID
<code>gid</code>	組 ID

這些選項對 `remount` 沒有任何影響。您可以通過 `chmod(1)`, `chown(1)` 和 `chgrp(1)` 的更改已經掛載的參數。

`tmpfs` 具有選擇 32 位還是 64 位 `inode` 的掛載選項：

inode64	使用 64 位 inode
inode32	使用 32 位 inode

在 32 位內核上，默認是 inode32，掛載時指定 inode64 會被拒絕。在 64 位內核上，默認配置是 CONFIG\_TMPFS\_INODE64。inode64 避免了單個設備上可能有多個具有相同 inode 編號的文件；比如 32 位應用程式使用 glibc 如果長期訪問 tmpfs，一旦達到 33 位 inode 編號，就有 EOVERRFLOW 失敗的危險，無法打開大於 2GiB 的文件，並返回 EINVAL。

所以' mount -t tmpfs -o size=10G,nr\_inodes=10k,mode=700 tmpfs /mytmpfs' 將在 /mytmpfs 上掛載 tmpfs 實例，分配只能由 root 用戶訪問的 10GB RAM/SWAP，可以有 10240 個 inode 的實例。

作者 Christoph Rohland <cr@sap.com>, 1.12.01

更新 Hugh Dickins, 4 June 2007

更新 KOSAKI Motohiro, 16 Mar 2010

更新 Chris Down, 13 July 2020

TODOList:

- driver-api/index
- core-api/index
- locking/index
- accounting/index
- block/index
- cdrom/index
- ide/index
- fb/index
- fpga/index
- hid/index
- i2c/index
- iio/index
- isdn/index
- infiniband/index
- leds/index
- netlabel/index
- networking/index
- pcmcia/index

- power/index
- target/index
- timers/index
- spi/index
- w1/index
- watchdog/index
- virt/index
- input/index
- hwmon/index
- gpu/index
- security/index
- sound/index
- crypto/index
- vm/index
- bpf/index
- usb/index
- PCI/index
- scsi/index
- misc-devices/index
- scheduler/index
- mhi/index

### \* 體系結構無關文檔

TODOList:

- asm-annotations

## \* 特定體系結構文檔

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

---

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:src.res@email.cn)。

---

**Original** Documentation/arm64/index.rst

**Translator** Bailu Lin <[bailu.lin@vivo.com](mailto:bailu.lin@vivo.com)> Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

## \* ARM64 架構

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

---

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:src.res@email.cn)。

---

**Original** Documentation/arm64/amu.rst

**Translator:** Bailu Lin <[bailu.lin@vivo.com](mailto:bailu.lin@vivo.com)> Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

## AArch64 Linux 中擴展的活動監控單元

作者: Ionela Voinescu <[ionela.voinescu@arm.com](mailto:ionela.voinescu@arm.com)>

日期: 2019-09-10

本文檔簡要描述了 AArch64 Linux 支持的活動監控單元的規範。

### 架構總述

活動監控是 ARMv8.4 CPU 架構引入的一個可選擴展特性。

活動監控單元 (在每個 CPU 中實現) 為系統管理提供了性能計數器。既可以通過系統寄存器的方式訪問計數器，同時也支持外部內存映射的方式訪問計數器。

AMUv1 架構實現了一個由 4 個固定的 64 位事件計數器組成的計數器組。

- CPU 周期計數器：同 CPU 的頻率增長
- 常量計數器：同固定的系統時鐘頻率增長
- 淘汰指令計數器：同每次架構指令執行增長
- 內存停頓周期計數器：計算由在時鐘域內的最後一級緩存中未命中而引起的指令調度停頓周期數

當處於 WFI 或者 WFE 狀態時，計數器不會增長。

AMU 架構提供了一個高達 16 位的事件計數器空間，未來新的 AMU 版本中可能用它來實現新增的事件計數器。

另外，AMUv1 實現了一個多達 16 個 64 位輔助事件計數器的計數器組。

冷復位時所有的計數器會清零。

### 基本支持

內核可以安全地運行在支持 AMU 和不支持 AMU 的 CPU 組合中。因此，當配置 CONFIG\_ARM64\_AMU\_EXTN 後我們無條件使能後續 (secondary or hotplugged) CPU 檢測和使用這個特性。

當在 CPU 上檢測到該特性時，我們會標記為特性可用但是不能保證計數器的功能，僅表明有擴展屬性。

固件 (代碼運行在高異常級別，例如 arm-tf ) 需支持以下功能：

- 提供低異常級別 (EL2 和 EL1) 訪問 AMU 寄存器的能力。
- 使能計數器。如果未使能，它的值應為 0。
- 在從電源關閉狀態啓動 CPU 前或後保存或者恢復計數器。

當使用使能了該特性的內核啓動但固件損壞時，訪問計數器寄存器可能會遭遇 panic 或者死鎖。即使未發現這些症狀，計數器寄存器返回的數據結果並不一定能反映真實情況。通常，計數器會返回 0，表明他們未被使能。

如果固件沒有提供適當的支持最好關閉 CONFIG\_ARM64\_AMU\_EXTN。值得注意的是，出於安全原因，不要繞過 AMUSERRENRL\_ELO 設置而捕獲從 EL0(用戶空間) 訪問 EL1(內核空間)。因此，固件應該確保訪問 AMU 寄存器不會困在 EL2 或 EL3。

AMUv1 的固定計數器可以通過如下系統寄存器訪問：

- SYS\_AMEVCNTR0\_CORE\_ELO

- SYS\_AMEVCNTR0\_CONST\_ELO
- SYS\_AMEVCNTR0\_INST\_RET\_ELO
- SYS\_AMEVCNTR0\_MEM\_STALL\_ELO

特定輔助計數器可以通過 SYS\_AMEVCNTR1\_EL0(n) 訪問，其中 n 介於 0 到 15。

詳細信息定義在目錄：arch/arm64/include/asm/sysreg.h。

## 用戶空間訪問

由於以下原因，當前禁止從用戶空間訪問 AMU 的寄存器：

- 安全因數：可能會暴露處於安全模式執行的代碼信息。
- 意願：AMU 是用於系統管理的。

同樣，該功能對用戶空間不可見。

## 虛擬化

由於以下原因，當前禁止從 KVM 客戶端的用戶空間 (EL0) 和內核空間 (EL1) 訪問 AMU 的寄存器：

- 安全因數：可能會暴露給其他客戶端或主機端執行的代碼信息。

任何試圖訪問 AMU 寄存器的行爲都會觸發一個註冊在客戶端的未定義異常。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

**Original** Documentation/arm64/hugetlbpage.rst

**Translator:** Bailu Lin <[bailu.lin@vivo.com](mailto:bailu.lin@vivo.com)> Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

### ARM64 中的 HugeTLBpage

大頁依靠有效利用 TLBs 來提高地址翻譯的性能。這取決於以下兩點 -

- 大頁的大小
- TLBs 支持的條目大小

ARM64 接口支持 2 種大頁方式。

#### 1) pud/pmd 級別的塊映射

這是常規大頁，他們的 pmd 或 pud 頁面表條目指向一個內存塊。不管 TLB 中支持的條目大小如何，塊映射可以減少翻譯大頁地址所需遍歷的頁表深度。

#### 2) 使用連續位

架構中轉換頁表條目 (D4.5.3, ARM DDI 0487C.a) 中提供一個連續位告訴 MMU 這個條目是一個連續條目集的一員，它可以被緩存在單個 TLB 條目中。

在 Linux 中連續位用來增加 pmd 和 pte(最後一級) 級別映射的大小。受支持的連續頁表條目數量因頁面大小和頁表級別而異。

支持以下大頁尺寸配置 -

•	CONT PTE	PMD	CONT PMD	PUD
4K:	64K	2M	32M	1G
16K:	2M	32M	1G	
64K:	2M	512M	16G	

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

Original Documentation/arm64/perf.rst

Translator: Bailu Lin <[bailu.lin@vivo.com](mailto:bailu.lin@vivo.com)> Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

## Perf 事件屬性

作者 Andrew Murray <[andrew.murray@arm.com](mailto:andrew.murray@arm.com)>

日期 2019-03-06

### **exclude\_user**

該屬性排除用戶空間。

用戶空間始終運行在 EL0，因此該屬性將排除 EL0。

### **exclude\_kernel**

該屬性排除內核空間。

打開 VHE 時內核運行在 EL2，不打開 VHE 時內核運行在 EL1。客戶機內核總是運行在 EL1。

對於宿主機，該屬性排除 EL1 和 VHE 上的 EL2。

對於客戶機，該屬性排除 EL1。請注意客戶機從來不會運行在 EL2。

### **exclude\_hv**

該屬性排除虛擬機監控器。

對於 VHE 宿主機該屬性將被忽略，此時我們認為宿主機內核是虛擬機監控器。

對於 non-VHE 宿主機該屬性將排除 EL2，因為虛擬機監控器運行在 EL2 的任何代碼主要用於客戶機和宿主機的切換。

對於客戶機該屬性無效。請注意客戶機從來不會運行在 EL2。

### **exclude\_host / exclude\_guest**

這些屬性分別排除了 KVM 宿主機和客戶機。

KVM 宿主機可能運行在 EL0（用戶空間），EL1（non-VHE 內核）和 EL2（VHE 內核或 non-VHE 虛擬機監控器）。

KVM 客戶機可能運行在 EL0（用戶空間）和 EL1（內核）。

由於宿主機和客戶機之間重疊的異常級別，我們不能僅僅依靠 PMU 的硬體異常過濾機制-因此我們必須啓用/禁用對於客戶機進入和退出的計數。而這在 VHE 和 non-VHE 系統上表現不同。

對於 non-VHE 系統的 `exclude_host` 屬性排除 EL2 - 在進入和退出客戶機時，我們會根據 `exclude_host` 和 `exclude_guest` 屬性在適當的情況下禁用/啓用該事件。

對於 VHE 系統的 exclude\_guest 屬性排除 EL1，而對其中的 exclude\_host 屬性同時排除 EL0，EL2。在進入和退出客戶機時，我們會適當地根據 exclude\_host 和 exclude\_guest 屬性包括/排除 EL0。

以上聲明也適用於在 not-VHE 客戶機使用這些屬性時，但是請注意客戶機從來不會運行在 EL2。

### 準確性

在 non-VHE 宿主機上，我們在 EL2 進入/退出宿主機/客戶機的切換時啓用/關閉計數器 -但是在啓用/禁用計數器和進入/退出客戶機之間存在一段延時。對於 exclude\_host，我們可以通過過濾 EL2 消除在客戶機進入/退出邊界上用於計數客戶機事件的宿主機事件計數器。但是當使用!exclude\_hv 時，在客戶機進入/退出有一個小的停電窗口無法捕獲到宿主機的事件。

在 VHE 系統沒有停電窗口。

**Warning:** 此文件的目的是為讓中文讀者更容易閱讀和理解，而不是作為一個分支。因此，如果您對此文件有任何意見或改動，請先嘗試更新原始英文文件。如果要更改或修正某處翻譯文件，請將意見或補丁發送給維護者（聯繫方式見下）。

**Note:** 如果您發現本文檔與原始文件有任何不同或者有翻譯問題，請聯繫該文件的譯者，或者發送電子郵件給胡皓文以獲取幫助：[<src.res@email.cn>](mailto:<src.res@email.cn>)。

**Original** Documentation/arm64/elf\_hwcaps.rst

**Translator:** Bailu Lin <[bailu.lin@vivo.com](mailto:bailu.lin@vivo.com)> Hu Haowen <[src.res@email.cn](mailto:src.res@email.cn)>

### ARM64 ELF hwcaps

這篇文檔描述了 arm64 ELF hwcaps 的用法和語義。

#### 1. 簡介

有些硬體或軟體功能僅在某些 CPU 實現上和/或在具體某個內核配置上可用，但對於處於 EL0 的用戶空間代碼沒有可用的架構發現機制。內核通過在輔助向量表公開一組稱為 hwcaps 的標誌而把這些功能暴露給用戶空間。

用戶空間軟體可以通過獲取輔助向量的 AT\_HWCAP 或 AT\_HWCAP2 條目來測試功能，並測試是否設置了相關標誌，例如：

```
bool floating_point_is_present(void)
{
    unsigned long hwcaps = getauxval(AT_HWCAP);
    if (hwcaps & HWCAP_FP)
```

```

        return true;

    return false;
}

```

如果軟體依賴於 hwcap 描述的功能，在嘗試使用該功能前則應檢查相關的 hwcap 標誌以驗證該功能是否存在。

不能通過其他方式探查這些功能。當一個功能不可用時，嘗試使用它可能導致不可預測的行為，並且無法保證能確切的知道該功能不可用，例如 SIGILL。

## 2. Hwcaps 的說明

大多數 hwcaps 旨在說明通過架構 ID 寄存器 (處於 EL0 的用戶空間代碼無法訪問) 描述的功能的存在。這些 hwcap 通過 ID 寄存器欄位定義，並且應根據 ARM 體系結構參考手冊 (ARM ARM) 中定義的欄位來解釋說明。

這些 hwcaps 以下面的形式描述：

`idreg.field == val` 表示有某個功能。

當 `idreg.field` 中有 `val` 時，hwcaps 表示 ARM ARM 定義的功能是有效的，但是並不是說要完全和 `val` 相等，也不是說 `idreg.field` 描述的其他功能就是缺失的。

其他 hwcaps 可能表明無法僅由 ID 寄存器描述的功能的存在。這些 hwcaps 可能沒有被 ID 寄存器描述，需要參考其他文檔。

## 3. AT\_HWCAP 中揭示的 hwcaps

**HWCAP\_FP** `ID_AA64PFR0_EL1.FP == 0b0000` 表示有此功能。

**HWCAP\_ASIMD** `ID_AA64PFR0_EL1.AdvSIMD == 0b0000` 表示有此功能。

**HWCAP\_EVTSTRM** 通用計時器頻率配置為大約 100KHz 以生成事件。

**HWCAP\_AES** `ID_AA64ISAR0_EL1.AES == 0b0001` 表示有此功能。

**HWCAP\_PMULL** `ID_AA64ISAR0_EL1.AES == 0b0010` 表示有此功能。

**HWCAP\_SHA1** `ID_AA64ISAR0_EL1.SHA1 == 0b0001` 表示有此功能。

**HWCAP\_SHA2** `ID_AA64ISAR0_EL1.SHA2 == 0b0001` 表示有此功能。

**HWCAP\_CRC32** `ID_AA64ISAR0_EL1.CRC32 == 0b0001` 表示有此功能。

**HWCAP\_ATOMICS** `ID_AA64ISAR0_EL1.Atomic == 0b0010` 表示有此功能。

**HWCAP\_FPHP** `ID_AA64PFR0_EL1.FP == 0b0001` 表示有此功能。

**HWCAP\_ASIMDHP** `ID_AA64PFR0_EL1.AdvSIMD == 0b0001` 表示有此功能。

**HWCAP\_CPUID** 根據 Documentation/arm64/cpu-feature-registers.rst 描述，EL0 可以訪問某些 ID 寄存器。

這些 ID 寄存器可能表示功能的可用性。

**HWCAP\_ASIMDRDM** ID\_AA64ISAR0\_EL1.RDM == 0b0001 表示有此功能。

**HWCAP\_JSCVT** ID\_AA64ISAR1\_EL1.JSCVT == 0b0001 表示有此功能。

**HWCAP\_FCMA** ID\_AA64ISAR1\_EL1.FCMA == 0b0001 表示有此功能。

**HWCAP\_LRCPC** ID\_AA64ISAR1\_EL1.LRCPC == 0b0001 表示有此功能。

**HWCAP\_DCPOP** ID\_AA64ISAR1\_EL1.DPB == 0b0001 表示有此功能。

**HWCAP\_SHA3** ID\_AA64ISAR0\_EL1.SHA3 == 0b0001 表示有此功能。

**HWCAP\_SM3** ID\_AA64ISAR0\_EL1.SM3 == 0b0001 表示有此功能。

**HWCAP\_SM4** ID\_AA64ISAR0\_EL1.SM4 == 0b0001 表示有此功能。

**HWCAP\_ASIMDDP** ID\_AA64ISAR0\_EL1.DP == 0b0001 表示有此功能。

**HWCAP\_SHA512** ID\_AA64ISAR0\_EL1.SHA2 == 0b0010 表示有此功能。

**HWCAP\_SVE** ID\_AA64PFR0\_EL1.SVE == 0b0001 表示有此功能。

**HWCAP\_ASIMDFHM** ID\_AA64ISAR0\_EL1.FHM == 0b0001 表示有此功能。

**HWCAP\_DIT** ID\_AA64PFR0\_EL1.DIT == 0b0001 表示有此功能。

**HWCAP\_USCAT** ID\_AA64MMFR2\_EL1.AT == 0b0001 表示有此功能。

**HWCAP\_ILRCPC** ID\_AA64ISAR1\_EL1.LRCPC == 0b0010 表示有此功能。

**HWCAP\_FLAGM** ID\_AA64ISAR0\_EL1.TS == 0b0001 表示有此功能。

**HWCAP\_SSBS** ID\_AA64PFR1\_EL1.SSBS == 0b0010 表示有此功能。

**HWCAP\_SB** ID\_AA64ISAR1\_EL1.SB == 0b0001 表示有此功能。

**HWCAP\_PACA** 如 Documentation/arm64/pointer-authentication.rst 所描述，  
ID\_AA64ISAR1\_EL1.APA == 0b0001 或 ID\_AA64ISAR1\_EL1.API == 0b0001 表示有此  
功能。

**HWCAP\_PACG** 如 Documentation/arm64/pointer-authentication.rst 所描述，  
ID\_AA64ISAR1\_EL1.GPA == 0b0001 或 ID\_AA64ISAR1\_EL1.GPI == 0b0001 表示有此  
功能。

**HWCAP2\_DCPODP**

ID\_AA64ISAR1\_EL1.DPB == 0b0010 表示有此功能。

**HWCAP2\_SVE2**

ID\_AA64ZFR0\_EL1.SVEVer == 0b0001 表示有此功能。

**HWCAP2\_SVEAES**

ID\_AA64ZFR0\_EL1.AES == 0b0001 表示有此功能。

#### HWCAP2\_SVEPMULL

ID\_AA64ZFR0\_EL1.AES == 0b0010 表示有此功能。

#### HWCAP2\_SVEBITPERM

ID\_AA64ZFR0\_EL1.BitPerm == 0b0001 表示有此功能。

#### HWCAP2\_SVESHA3

ID\_AA64ZFR0\_EL1.SHA3 == 0b0001 表示有此功能。

#### HWCAP2\_SVESM4

ID\_AA64ZFR0\_EL1.SM4 == 0b0001 表示有此功能。

#### HWCAP2\_FLAGM2

ID\_AA64ISAR0\_EL1.TS == 0b0010 表示有此功能。

#### HWCAP2\_FRINT

ID\_AA64ISAR1\_EL1.FRINTTS == 0b0001 表示有此功能。

#### HWCAP2\_SVEI8MM

ID\_AA64ZFR0\_EL1.I8MM == 0b0001 表示有此功能。

#### HWCAP2\_SVEF32MM

ID\_AA64ZFR0\_EL1.F32MM == 0b0001 表示有此功能。

#### HWCAP2\_SVEF64MM

ID\_AA64ZFR0\_EL1.F64MM == 0b0001 表示有此功能。

#### HWCAP2\_SVEBF16

ID\_AA64ZFR0\_EL1.BF16 == 0b0001 表示有此功能。

#### HWCAP2\_I8MM

ID\_AA64ISAR1\_EL1.I8MM == 0b0001 表示有此功能。

#### HWCAP2\_BF16

ID\_AA64ISAR1\_EL1.BF16 == 0b0001 表示有此功能。

#### HWCAP2\_DGH

ID\_AA64ISAR1\_EL1.DGH == 0b0001 表示有此功能。

#### HWCAP2\_RNG

ID\_AA64ISAR0\_EL1.RNDR == 0b0001 表示有此功能。

#### HWCAP2\_BTI

ID\_AA64PFR0\_EL1.BT == 0b0001 表示有此功能。

### 4. 未使用的 AT\_HWCAP 位

為了與用戶空間交互，內核保證 AT\_HWCAP 的第 62、63 位將始終返回 0。

TODOList:

- arch

### \* 其他文檔

有幾份未排序的文檔似乎不適合放在文檔的其他部分，或者可能需要進行一些調整和/或轉換為 reStructuredText 格式，也有可能太舊。

TODOList:

- staging/index
- watch\_queue

### \* 目錄和表格

- genindex

## TRADUZIONE ITALIANA

**manutentore** Federico Vaga <[federico.vaga@vaga.pv.it](mailto:federico.vaga@vaga.pv.it)>

### \* Avvertenze

L'obiettivo di questa traduzione è di rendere più facile la lettura e la comprensione per chi non capisce l'inglese o ha dubbi sulla sua interpretazione, oppure semplicemente per chi preferisce leggere in lingua italiana. Tuttavia, tenete ben presente che l'*unica* documentazione ufficiale è quella in lingua inglese: `linux_doc`

La propagazione simultanea a tutte le traduzioni di una modifica in `linux_doc` è altamente improbabile. I manutentori delle traduzioni - e i contributori - seguono l'evolversi della documentazione ufficiale e cercano di mantenere le rispettive traduzioni allineate nel limite del possibile. Per questo motivo non c'è garanzia che una traduzione sia aggiornata all'ultima modifica. Se quello che leggete in una traduzione non corrisponde a quello che leggete nel codice, informate il manutentore della traduzione e - se potete - verificate anche la documentazione in inglese.

Una traduzione non è un *fork* della documentazione ufficiale, perciò gli utenti non vi troveranno alcuna informazione diversa rispetto alla versione ufficiale. Ogni aggiunta, rimozione o modifica dei contenuti deve essere fatta prima nei documenti in inglese. In seguito, e quando è possibile, la stessa modifica dovrebbe essere applicata anche alle traduzioni. I manutentori delle traduzioni accettano contributi che interessano prettamente l'attività di traduzione (per esempio, nuove traduzioni, aggiornamenti, correzioni).

Le traduzioni cercano di essere il più possibile accurate ma non è possibile mappare direttamente una lingua in un'altra. Ogni lingua ha la sua grammatica e una sua cultura alle spalle, quindi la traduzione di una frase in inglese potrebbe essere modificata per adattarla all'italiano. Per questo motivo, quando leggete questa traduzione, potrete trovare alcune differenze di forma ma che trasmettono comunque il messaggio originale. Nonostante la grande diffusione di ingleismi nella lingua parlata, quando possibile, questi verranno sostituiti dalle corrispondenti parole italiane

Se avete bisogno d'aiuto per comunicare con la comunità Linux ma non vi sentite a vostro agio nello scrivere in inglese, potete chiedere aiuto al manutentore della traduzione.

## \* La documentazione del kernel Linux

Questo è il livello principale della documentazione del kernel in lingua italiana. La traduzione è incompleta, noterete degli avvisi che vi segnaleranno la mancanza di una traduzione o di un gruppo di traduzioni.

Più in generale, la documentazione, come il kernel stesso, sono in costante sviluppo; particolarmente vero in quanto stiamo lavorando alla riorganizzazione della documentazione in modo più coerente. I miglioramenti alla documentazione sono sempre i benvenuti; per cui, se vuoi aiutare, iscriviti alla lista di discussione linux-doc presso vger.kernel.org.

## \* Documentazione sulla licenza dei sorgenti

I seguenti documenti descrivono la licenza usata nei sorgenti del kernel Linux (GPLv2), come licenziare i singoli file; inoltre troverete i riferimenti al testo integrale della licenza.

- [it\\_kernel\\_licensing](#)

## \* Documentazione per gli utenti

I seguenti manuali sono scritti per gli *utenti* del kernel - ovvero, coloro che cercano di farlo funzionare in modo ottimale su un dato sistema

**Warning:** TODO ancora da tradurre

## \* Documentazione per gli sviluppatori di applicazioni

Il manuale delle API verso lo spazio utente è una collezione di documenti che descrivono le interfacce del kernel viste dagli sviluppatori di applicazioni.

**Warning:** TODO ancora da tradurre

## \* Introduzione allo sviluppo del kernel

Questi manuali contengono informazioni su come contribuire allo sviluppo del kernel. Attorno al kernel Linux gira una comunità molto grande con migliaia di sviluppatori che contribuiscono ogni anno. Come in ogni grande comunità, sapere come le cose vengono fatte renderà il processo di integrazione delle vostre modifiche molto più semplice

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/index.rst

**Translator** Federico Vaga <[federico.vaga@vaga.pv.it](mailto:federico.vaga@vaga.pv.it)>

## Lavorare con la comunità di sviluppo del kernel

Quindi volete diventare sviluppatori del kernel? Benvenuti! C'è molto da imparare sul lato tecnico del kernel, ma è anche importante capire come funziona la nostra comunità. Leggere questi documenti renderà più facile l'accettazione delle vostre modifiche con il minimo sforzo.

Di seguito le guide che ogni sviluppatore dovrebbe leggere.

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/howto.rst

**Translator** Alessia Mantegazza <[amantegazza@vaga.pv.it](mailto:amantegazza@vaga.pv.it)>

## Come partecipare allo sviluppo del kernel Linux

Questo è il documento fulcro di quanto trattato sull'argomento. Esso contiene le istruzioni su come diventare uno sviluppatore del kernel Linux e spiega come lavorare con la comunità di sviluppo kernel Linux. Il documento non tratterà alcun aspetto tecnico relativo alla programmazione del kernel, ma vi aiuterà indirizzandovi sulla corretta strada.

Se qualsiasi cosa presente in questo documento diventasse obsoleta, vi preghiamo di inviare le correzioni agli amministratori di questo file, indicati in fondo al presente documento.

## Introduzione

Dunque, volete imparare come diventare sviluppatori del kernel Linux? O vi è stato detto dal vostro capo, "Vai, scrivi un driver Linux per questo dispositivo". Bene, l'obbiettivo di questo documento è quello di insegnarvi tutto ciò che dovete sapere per raggiungere il vostro scopo descrivendo il procedimento da seguire e consigliandovi su come lavorare con la comunità. Il documento cercherà, inoltre, di spiegare alcune delle ragioni per le quali la comunità lavora in un modo suo particolare.

Il kernel è scritto prevalentemente nel linguaggio C con alcune parti specifiche dell'architettura scritte in linguaggio assembly. Per lo sviluppo kernel è richiesta una buona conoscenza del linguaggio C. L'assembly (di qualsiasi architettura) non è richiesto, a meno che non pensiate di fare dello sviluppo di basso livello per un'architettura. Sebbene essi non siano un buon sostituto ad un solido studio del linguaggio C o ad anni di esperienza, i seguenti libri sono, se non altro, utili riferimenti:

- "The C Programming Language" di Kernighan e Ritchie [Prentice Hall]
- "Practical C Programming" di Steve Oualline [O'Reilly]
- "C: A Reference Manual" di Harbison and Steele [Prentice Hall]

Il kernel è stato scritto usando GNU C e la toolchain GNU. Sebbene si attenga allo standard ISO C89, esso utilizza una serie di estensioni che non sono previste in questo standard. Il kernel è un ambiente C indipendente, che non ha alcuna dipendenza dalle librerie C standard, così alcune parti del C standard non sono supportate. Le divisioni long long e numeri in virgola mobile non sono permessi. Qualche volta è difficile comprendere gli assunti che il kernel ha

riguardo gli strumenti e le estensioni in uso, e sfortunatamente non esiste alcuna indicazione definitiva. Per maggiori informazioni, controllate, la pagina *info gcc*.

Tenete a mente che state cercando di apprendere come lavorare con la comunità di sviluppo già esistente. Questo è un gruppo eterogeneo di persone, con alti standard di codifica, di stile e di procedura. Questi standard sono stati creati nel corso del tempo basandosi su quanto hanno riscontrato funzionare al meglio per un squadra così grande e geograficamente sparsa. Cercate di imparare, in anticipo, il più possibile circa questi standard, poichè ben spiegati; non aspettatevi che gli altri si adattino al vostro modo di fare o a quello della vostra azienda.

### Note legali

Il codice sorgente del kernel Linux è rilasciato sotto GPL. Siete pregati di visionare il file, COPYING, presente nella cartella principale dei sorgente, per eventuali dettagli sulla licenza. Se avete ulteriori domande sulla licenza, contattate un avvocato, non chiedete sulle liste di discussione del kernel Linux. Le persone presenti in queste liste non sono avvocati, e non dovreste basarvi sulle loro dichiarazioni in materia giuridica.

Per domande più frequenti e risposte sulla licenza GPL, guardare:

<https://www.gnu.org/licenses/gpl-faq.html>

### Documentazione

I sorgenti del kernel Linux hanno una vasta base di documenti che vi insegheranno come interagire con la comunità del kernel. Quando nuove funzionalità vengono aggiunte al kernel, si raccomanda di aggiungere anche i relativi file di documentation che spiegano come usarele. Quando un cambiamento del kernel genera anche un cambiamento nell'interfaccia con lo spazio utente, è raccomandabile che inviate una notifica o una correzione alle pagine *man* spiegando tale modifica agli amministratori di queste pagine all'indirizzo [mtk.manpages@gmail.com](mailto:mtk.manpages@gmail.com), aggiungendo in CC la lista [linux-api@vger.kernel.org](mailto:linux-api@vger.kernel.org).

Di seguito una lista di file che sono presenti nei sorgenti del kernel e che è richiesto che voi leggiate:

**Documentation/translations/it\_IT/admin-guide/README.rst** Questo file da una piccola anteprima del kernel Linux e descrive il minimo necessario per configurare e generare il kernel. I novizi del kernel dovrebbero iniziare da qui.

*Documentation/translations/it\_IT/process/changes.rst*

Questo file fornisce una lista dei pacchetti software necessari a compilare e far funzionare il kernel con successo.

*Documentation/translations/it\_IT/process/coding-style.rst*

Questo file descrive lo stile della codifica per il kernel Linux, e parte delle motivazioni che ne sono alla base. Tutto il nuovo codice deve seguire le linee guida in questo documento. Molti amministratori accetteranno patch solo se queste osserveranno tali regole, e molte persone revisioneranno il codice solo se scritto nello stile appropriato.

*Documentation/translations/it\_IT/process/submitting-patches.rst*

*Documentation/translations/it\_IT/process/submitting-drivers.rst*

e

Questo file descrive dettagliatamente come creare ed inviare una patch con successo, includendo (ma non solo questo):

- Contenuto delle email
- Formato delle email
- I destinatari delle email

Seguire tali regole non garantirà il successo (tutte le patch sono soggette a controlli realitivi a contenuto e stile), ma non seguirle lo precluderà sempre.

Altre ottime descrizioni di come creare buone patch sono:

**“The Perfect Patch”** <https://www.ozlabs.org/~akpm/stuff/tpp.txt>

**“Linux kernel patch submission format”** <https://web.archive.org/web/20180829112450/http://linux.yyz.us/patch-format.html>

*Documentation/translations/it\_IT/process/stable-api-nonsense.rst*

Questo file descrive la motivazioni sottostanti la conscia decisione di non avere un API stabile all'interno del kernel, incluso cose come:

- Sottosistemi shim-layers (per compatibilità?)
- Portabilità fra Sistemi Operativi dei driver.
- Attenuare i rapidi cambiamenti all'interno dei sorgenti del kernel (o prevenirli)

Questo documento è vitale per la comprensione della filosofia alla base dello sviluppo di Linux ed è molto importante per le persone che arrivano da esperienze con altri Sistemi Operativi.

**Documentation/translations/it\_IT/admin-guide/security-bugs.rst** Se ritenete di aver trovato un problema di sicurezza nel kernel Linux, seguite i passaggi scritti in questo documento per notificarlo agli sviluppatori del kernel, ed aiutare la risoluzione del problema.

**Documentation/translations/it\_IT/process/management-style.rst** Questo documento descrive come i manutentori del kernel Linux operano e la filosofia comune alla base del loro metodo. Questa è un'importante lettura per tutti coloro che sono nuovi allo sviluppo del kernel (o per chi è semplicemente curioso), poiché risolve molti dei più comuni fraintendimenti e confusioni dovuti al particolare comportamento dei manutentori del kernel.

**Documentation/translations/it\_IT/process/stable-kernel-rules.rst** Questo file descrive le regole sulle quali vengono basati i rilasci del kernel, e spiega cosa fare se si vuole che una modifica venga inserita in uno di questi rilasci.

**Documentation/translations/it\_IT/process/kernel-docs.rst** Una lista di documenti pertinenti allo sviluppo del kernel. Per favore consultate questa lista se non trovate ciò che cercate nella documentazione interna del kernel.

**Documentation/translations/it\_IT/process/applying-patches.rst** Una buona introduzione che descrivere esattamente cos'è una patch e come applicarla ai differenti rami di sviluppo del kernel.

Il kernel inoltre ha un vasto numero di documenti che possono essere automaticamente generati dal codice sorgente stesso o da file ReStructuredText (ReST), come questo. Esso include una

completa descrizione dell'API interna del kernel, e le regole su come gestire la sincronizzazione (locking) correttamente

Tutte queste tipologie di documenti possono essere generati in PDF o in HTML utilizzando:

```
make pdfdocs  
make htmldocs
```

rispettivamente dalla cartella principale dei sorgenti del kernel.

I documenti che impiegano ReST saranno generati nella cartella Documentation/output. Questi possono essere generati anche in formato LaTeX e ePub con:

```
make latexdocs  
make epubdocs
```

## Diventare uno sviluppatore del kernel

Se non sapete nulla sullo sviluppo del kernel Linux, dovreste dare uno sguardo al progetto *Linux KernelNewbies*:

<https://kernelnewbies.org>

Esso prevede un'utile lista di discussione dove potete porre più o meno ogni tipo di quesito relativo ai concetti fondamentali sullo sviluppo del kernel (assicuratevi di cercare negli archivi, prima di chiedere qualcosa alla quale è già stata fornita risposta in passato). Esistono inoltre, un canale IRC che potete usare per formulare domande in tempo reale, e molti documenti utili che vi faciliteranno nell'apprendimento dello sviluppo del kernel Linux.

Il sito internet contiene informazioni di base circa l'organizzazione del codice, sottosistemi e progetti attuali (sia interni che esterni a Linux). Esso descrive, inoltre, informazioni logistiche di base, riguardanti ad esempio la compilazione del kernel e l'applicazione di una modifica.

Se non sapete dove cominciare, ma volete cercare delle attività dalle quali partire per partecipare alla comunità di sviluppo, andate al progetto Linux Kernel Janitor's.

<https://kernelnewbies.org/KernelJanitors>

È un buon posto da cui iniziare. Esso presenta una lista di problematiche relativamente semplici da sistemare e pulire all'interno della sorgente del kernel Linux. Lavorando con gli sviluppatori incaricati di questo progetto, imparerete le basi per l'inserimento delle vostre modifiche all'interno dei sorgenti del kernel Linux, e possibilmente, sarete indirizzati al lavoro successivo da svolgere, se non ne avrete ancora idea.

Prima di apportare una qualsiasi modifica al codice del kernel Linux, è imperativo comprendere come tale codice funziona. A questo scopo, non c'è nulla di meglio che leggerlo direttamente (la maggior parte dei bit più complessi sono ben commentati), eventualmente anche con l'aiuto di strumenti specializzati. Uno degli strumenti che è particolarmente raccomandato è il progetto Linux Cross-Reference, che è in grado di presentare codice sorgente in un formato autoreferenziale ed indicizzato. Un eccellente ed aggiornata fonte di consultazione del codice del kernel la potete trovare qui:

<https://elixir.bootlin.com/>

## Il processo di sviluppo

Il processo di sviluppo del kernel Linux si compone di pochi “rami” principali e di molti altri rami per specifici sottosistemi. Questi rami sono:

- I sorgenti kernel 4.x
- I sorgenti stabili del kernel 4.x.y -stable
- Sorgenti dei sottosistemi del kernel e le loro modifiche
- Il kernel 4.x -next per test d'integrazione

### I sorgenti kernel 4.x

I kernel 4.x sono amministrati da Linus Torvald, e possono essere trovati su <https://kernel.org> nella cartella pub/linux/kernel/v4.x/. Il processo di sviluppo è il seguente:

- Non appena un nuovo kernel viene rilasciato si apre una finestra di due settimane. Durante questo periodo i manutentori possono proporre a Linus dei grossi cambiamenti; solitamente i cambiamenti che sono già stati inseriti nel ramo -next del kernel per alcune settimane. Il modo migliore per sottoporre dei cambiamenti è attraverso git (lo strumento usato per gestire i sorgenti del kernel, più informazioni sul sito <https://git-scm.com/>) ma anche delle patch vanno bene.
- Al termine delle due settimane un kernel -rc1 viene rilasciato e l'obiettivo ora è quello di renderlo il più solido possibile. A questo punto la maggior parte delle patch dovrebbero correggere un'eventuale regressione. I bachi che sono sempre esistiti non sono considerabili come regressioni, quindi inviate questo tipo di cambiamenti solo se sono importanti. Notate che un intero driver (o filesystem) potrebbe essere accettato dopo la -rc1 poiché non esistono rischi di una possibile regressione con tale cambiamento, fintanto che quest'ultimo è auto-contenuto e non influisce su aree esterne al codice che è stato aggiunto. git può essere utilizzato per inviare le patch a Linus dopo che la -rc1 è stata rilasciata, ma è anche necessario inviare le patch ad una lista di discussione pubblica per un'ulteriore revisione.
- Una nuova -rc viene rilasciata ogni volta che Linus reputa che gli attuali sorgenti siano in uno stato di salute ragionevolmente adeguato ai test. L'obiettivo è quello di rilasciare una nuova -rc ogni settimana.
- Il processo continua fino a che il kernel è considerato “pronto”; tale processo dovrebbe durare circa in 6 settimane.

È utile menzionare quanto scritto da Andrew Morton sulla lista di discussione kernel-linux in merito ai rilasci del kernel:

*“Nessuno sa quando un kernel verrà rilasciato, poichè questo è legato allo stato dei bachi e non ad una cronologia preventiva.”*

### I sorgenti stabili del kernel 4.x.y -stable

I kernel con versioni in 3-parti sono “kernel stabili”. Essi contengono correzioni critiche relativamente piccole nell’ambito della sicurezza oppure significative regressioni scoperte in un dato 4.x kernel.

Questo è il ramo raccomandato per gli utenti che vogliono un kernel recente e stabile e non sono interessati a dare il proprio contributo alla verifica delle versioni di sviluppo o sperimentali.

Se non è disponibile alcun kernel 4.x.y., quello più aggiornato e stabile sarà il kernel 4.x con la numerazione più alta.

4.x.y sono amministrati dal gruppo “stable” <[stable@vger.kernel.org](mailto:stable@vger.kernel.org)>, e sono rilasciati a seconda delle esigenze. Il normale periodo di rilascio è approssimativamente di due settimane, ma può essere più lungo se non si verificano problematiche urgenti. Un problema relativo alla sicurezza, invece, può determinare un rilascio immediato.

Il file Documentation/process/stable-kernel-rules.rst (nei sorgenti) documenta quali tipologie di modifiche sono accettate per i sorgenti -stable, e come avviene il processo di rilascio.

### Sorgenti dei sottosistemi del kernel e le loro patch

I manutentori dei diversi sottosistemi del kernel — ed anche molti sviluppatori di sottosistemi — mostrano il loro attuale stato di sviluppo nei loro repository. In questo modo, altri possono vedere cosa succede nelle diverse parti del kernel. In aree dove lo sviluppo è rapido, potrebbe essere chiesto ad uno sviluppatore di basare le proprie modifiche su questi repository in modo da evitare i conflitti fra le sottomissioni ed altri lavori in corso

La maggior parte di questi repository sono git, ma esistono anche altri SCM in uso, o file di patch pubblicate come una serie quilt. Gli indirizzi dei repository di sottosistema sono indicati nel file MAINTAINERS. Molti di questi possono essere trovati su <https://git.kernel.org/>.

Prima che una modifica venga inclusa in questi sottosistemi, sarà soggetta ad una revisione che inizialmente avviene tramite liste di discussione (vedere la sezione dedicata qui sotto). Per molti sottosistemi del kernel, tale processo di revisione è monitorato con lo strumento patchwork. Patchwork offre un’interfaccia web che mostra le patch pubblicate, inclusi i commenti o le revisioni fatte, e gli amministratori possono indicare le patch come “in revisione”, “accettate”, o “rifiutate”. Diversi siti Patchwork sono elencati al sito <https://patchwork.kernel.org/>.

### Il kernel 4.x -next per test d’integrazione

Prima che gli aggiornamenti dei sottosistemi siano accorpati nel ramo principale 4.x, sarà necessario un test d’integrazione. A tale scopo, esiste un repository speciale di test nel quale virtualmente tutti i rami dei sottosistemi vengono inclusi su base quotidiana:

<https://git.kernel.org/?p=linux/kernel/git/next/linux-next.git>

In questo modo, i kernel -next offrono uno sguardo riassuntivo su quello che ci si aspetterà essere nel kernel principale nel successivo periodo d’incorporazione. Coloro che vorranno fare dei test d’esecuzione del kernel -next sono più che benvenuti.

## Riportare Bug

Il file ‘Documentation/admin-guide/reporting-issues.rst’ nella cartella principale del kernel spiega come segnalare un baco nel kernel, e fornisce dettagli su quali informazioni sono necessarie agli sviluppatori del kernel per poter studiare il problema.

## Gestire i rapporti sui bug

Uno dei modi migliori per mettere in pratica le vostre capacità di hacking è quello di riparare bachi riportati da altre persone. Non solo aiuterete a far diventare il kernel più stabile, ma imparerete a riparare problemi veri dal mondo ed accrescerete le vostre competenze, e gli altri sviluppatori saranno al corrente della vostra presenza. Riparare bachi è una delle migliori vie per acquisire meriti tra gli altri sviluppatori, perché non a molte persone piace perdere tempo a sistemare i bachi di altri.

Per lavorare sui bachi già segnalati, per prima cosa cercate il sottosistema che vi interessa. Poi, verificate nel file MAINTAINERS dove vengono collezionati solitamente i bachi per quel sottosistema; spesso sarà una lista di discussione, raramente un bugtracker. Cercate bachi nell’archivio e aiutate dove credete di poterlo fare. Potete anche consultare <https://bugzilla.kernel.org>; però, solo una manciata di sottosistemi lo usano attivamente, ciò nonostante i bachi che coinvolgono l’intero kernel sono sempre riportati lì.

## Liste di discussione

Come descritto in molti dei documenti qui sopra, la maggior parte degli sviluppatori del kernel partecipano alla lista di discussione Linux Kernel. I dettagli su come iscriversi e disiscriversi dalla lista possono essere trovati al sito:

<http://vger.kernel.org/vger-lists.html#linux-kernel>

Ci sono diversi archivi della lista di discussione. Usate un qualsiasi motore di ricerca per trovarli. Per esempio:

<http://dir.gmane.org/gmane.linux.kernel>

È caldamente consigliata una ricerca in questi archivi sul tema che volete sollevare, prima di pubblicarlo sulla lista. Molte cose sono già state discusse in dettaglio e registrate negli archivi della lista di discussione.

Molti dei sottosistemi del kernel hanno anche una loro lista di discussione dedicata. Guardate nel file MAINTAINERS per avere una lista delle liste di discussione e il loro uso.

Molte di queste liste sono gestite su kernel.org. Per informazioni consultate la seguente pagina:

<http://vger.kernel.org/vger-lists.html>

Per favore ricordatevi della buona educazione quando utilizzate queste liste. Sebbene sia un po’ dozzinale, il seguente URL contiene alcune semplici linee guida per interagire con la lista (o con qualsiasi altra lista):

<http://www.albion.com/netiquette/>

Se diverse persone rispondono alla vostra mail, la lista dei riceventi (copia conoscenza) potrebbe diventare abbastanza lunga. Non cancellate nessuno dalla lista di CC: senza un buon motivo, e non rispondete solo all’indirizzo della lista di discussione. Fateci l’abitudine perché capita

spesso di ricevere la stessa email due volte: una dal mittente ed una dalla lista; e non cercate di modificarla aggiungendo intestazioni stravaganti, agli altri non piacerà.

Ricordate di rimanere sempre in argomento e di mantenere le attribuzioni delle vostre risposte invariate; mantenete il “John Kernelhacker wrote ...:” in cima alla vostra replica e aggiungete le vostre risposte fra i singoli blocchi citati, non scrivete all’inizio dell’email.

Se aggiungete patch alla vostra mail, assicuratevi che siano del tutto leggibili come indicato in Documentation/process/submitting-patches.rst. Gli sviluppatori kernel non vogliono avere a che fare con allegati o patch compresse; vogliono invece poter commentare le righe dei vostri cambiamenti, il che può funzionare solo in questo modo. Assicuratevi di utilizzare un gestore di mail che non alteri gli spazi ed i caratteri. Un ottimo primo test è quello di inviare a voi stessi una mail e cercare di sottoporre la vostra stessa patch. Se non funziona, sistemate il vostro programma di posta, o cambiatelo, finché non funziona.

Ed infine, per favore ricordatevi di mostrare rispetto per gli altri sottoscriventi.

### Lavorare con la comunità

L’obiettivo di questa comunità è quello di fornire il miglior kernel possibile. Quando inviate una modifica che volete integrare, sarà valutata esclusivamente dal punto di vista tecnico. Quindi, cosa dovreste aspettarvi?

- critiche
- commenti
- richieste di cambiamento
- richieste di spiegazioni
- nulla

Ricordatevi che questo fa parte dell’integrazione della vostra modifica all’interno del kernel. Dovete essere in grado di accettare le critiche, valutarle a livello tecnico ed eventualmente rielaborare nuovamente le vostre modifiche o fornire delle chiare e concise motivazioni per le quali le modifiche suggerite non dovrebbero essere fatte. Se non riceverete risposte, aspettate qualche giorno e riprovate ancora, qualche volta le cose si perdono nell’enorme mucchio di email.

Cosa non dovreste fare?

- aspettarvi che la vostra modifica venga accettata senza problemi
- mettervi sulla difensiva
- ignorare i commenti
- sottomettere nuovamente la modifica senza fare nessuno dei cambiamenti richiesti

In una comunità che è alla ricerca delle migliori soluzioni tecniche possibili, ci saranno sempre opinioni differenti sull’utilità di una modifica. Siate cooperativi e vogliate adattare la vostra idea in modo che sia inserita nel kernel. O almeno vogliate dimostrare che la vostra idea vale. Ricordatevi, sbagliare è accettato fintanto che siate disposti a lavorare verso una soluzione che è corretta.

È normale che le risposte alla vostra prima modifica possa essere semplicemente una lista con dozzine di cose che dovreste correggere. Questo **non** implica che la vostra patch non sarà

accettata, e questo **non** è contro di voi personalmente. Semplicemente correggete tutte le questioni sollevate contro la vostra modifica ed inviatela nuovamente.

## Differenze tra la comunità del kernel e le strutture aziendali

La comunità del kernel funziona diversamente rispetto a molti ambienti di sviluppo aziendali. Qui di seguito una lista di cose che potete provare a fare per evitare problemi:

Cose da dire riguardanti le modifiche da voi proposte:

- “Questo risolve più problematiche.”
- “Questo elimina 2000 stringhe di codice.”
- “Qui una modifica che spiega cosa sto cercando di fare.”
- “L’ho testato su 5 diverse architetture..”
- “Qui una serie di piccole modifiche che..”
- “Questo aumenta le prestazioni di macchine standard...”

Cose che dovreste evitare di dire:

- **“Lo abbiamo fatto in questo modo in AIX/ptx/Solaris, di conseguenza deve per forza essere giusto...”**
- “Ho fatto questo per 20 anni, quindi..”
- “Questo è richiesto dalla mia Azienda per far soldi”
- “Questo è per la linea di prodotti della nostra Azienda”
- **“Ecco il mio documento di design di 1000 pagine che descrive ciò che ho in mente”**
- “Ci ho lavorato per 6 mesi...”
- “Ecco una patch da 5000 righe che..”
- “Ho riscritto il pasticcio attuale, ed ecco qua..”
- “Ho una scadenza, e questa modifica ha bisogno di essere approvata ora”

Un’altra cosa nella quale la comunità del kernel si differenzia dai più classici ambienti di ingegneria del software è la natura “senza volto” delle interazioni umane. Uno dei benefici dell’uso delle email e di irc come forma primordiale di comunicazione è l’assenza di discriminazione basata su genere e razza. L’ambiente di lavoro Linux accetta donne e minoranze perché tutto quello che sei è un indirizzo email. Aiuta anche l’aspetto internazionale nel livellare il terreno di gioco perchè non è possibile indovinare il genere basandosi sul nome di una persona. Un uomo può chiamarsi Andrea ed una donna potrebbe chiamarsi Pat. Gran parte delle donne che hanno lavorato al kernel Linux e che hanno espresso una personale opinione hanno avuto esperienze positive.

La lingua potrebbe essere un ostacolo per quelle persone che non si trovano a loro agio con l’inglese. Una buona padronanza del linguaggio può essere necessaria per esporre le proprie idee in maniera appropriata all’interno delle liste di discussione, quindi è consigliabile che rilegiate le vostre email prima di inviarle in modo da essere certi che abbiano senso in inglese.

### Spezzare le vostre modifiche

La comunità del kernel Linux non accetta con piacere grossi pezzi di codice buttati lì tutti in una volta. Le modifiche necessitano di essere adeguatamente presentate, discusse, e suddivise in parti più piccole ed indipendenti. Questo è praticamente l'esatto opposto di quello che le aziende fanno solitamente. La vostra proposta dovrebbe, inoltre, essere presentata prestissimo nel processo di sviluppo, così che possiate ricevere un riscontro su quello che state facendo. Lasciate che la comunità senta che state lavorando con loro, e che non li stiate sfruttando come discarica per le vostre aggiunte. In ogni caso, non inviate 50 email nello stesso momento in una lista di discussione, il più delle volte la vostra serie di modifiche dovrebbe essere più piccola.

I motivi per i quali dovreste frammentare le cose sono i seguenti:

1) Piccole modifiche aumentano le probabilità che vengano accettate, altrimenti richiederebbe troppo tempo o sforzo nel verificarne la correttezza. Una modifica di 5 righe può essere accettata da un manutentore con a mala pena una seconda occhiata. Invece, una modifica da 500 linee può richiedere ore di rilettura per verificarne la correttezza (il tempo necessario è esponenzialmente proporzionale alla dimensione della modifica, o giù di lì)

Piccole modifiche sono inoltre molto facili da debuggare quando qualcosa non va. È molto più facile annullare le modifiche una per una che dissezionare una patch molto grande dopo la sua sottomissione (e rompere qualcosa).

2) È importante non solo inviare piccole modifiche, ma anche riscriverle e semplificarle (o più semplicemente ordinarle) prima di sottoporle.

Qui un'analogia dello sviluppatore kernel Al Viro:

*"Pensate ad un insegnante di matematica che corregge il compito di uno studente (di matematica). L'insegnante non vuole vedere le prove e gli errori commessi dallo studente prima che arrivi alla soluzione. Vuole vedere la risposta più pulita ed elegante possibile. Un buono studente lo sa, e non presenterebbe mai le proprie bozze prima prima della soluzione finale"*

*"Lo stesso vale per lo sviluppo del kernel. I manutentori ed i revisori non vogliono vedere il procedimento che sta dietro al problema che uno sta risolvendo. Vogliono vedere una soluzione semplice ed elegante."*

Può essere una vera sfida il saper mantenere l'equilibrio fra una presentazione elegante della vostra soluzione, lavorare insieme ad una comunità e dibattere su un lavoro incompleto. Pertanto è bene entrare presto nel processo di revisione per migliorare il vostro lavoro, ma anche per riuscire a tenere le vostre modifiche in pezzettini che potrebbero essere già accettate, nonostante la vostra intera attività non lo sia ancora.

In fine, rendetevi conto che non è accettabile inviare delle modifiche incomplete con la promessa che saranno "sistematiche dopo".

## Giustificare le vostre modifiche

Insieme alla frammentazione delle vostre modifiche, è altrettanto importante permettere alla comunità Linux di capire perché dovrebbero accettarle. Nuove funzionalità devono essere motivate come necessarie ed utili.

## Documentare le vostre modifiche

Quando inviate le vostre modifiche, fate particolare attenzione a quello che scrivete nella vostra email. Questa diventerà il *ChangeLog* per la modifica, e sarà visibile a tutti per sempre. Dovrebbe descrivere la modifica nella sua interezza, contenendo:

- perchè la modifica è necessaria
- l'approccio d'insieme alla patch
- dettagli supplementari
- risultati dei test

Per maggiori dettagli su come tutto ciò dovrebbe apparire, riferitevi alla sezione ChangeLog del documento:

**“The Perfect Patch”** <http://www.ozlabs.org/~akpm/stuff/tpp.txt>

A volte tutto questo è difficile da realizzare. Il perfezionamento di queste pratiche può richiedere anni (eventualmente). È un processo continuo di miglioramento che richiede molta pazienza e determinazione. Ma non mollate, si può fare. Molti lo hanno fatto prima, ed ognuno ha dovuto iniziare dove siete voi ora.

Grazie a Paolo Ciarrocchi che ha permesso che la sezione “Development Process” (<https://lwn.net/Articles/94386/>) fosse basata sui testi da lui scritti, ed a Randy Dunlap e Gerrit Huizenga per la lista di cose che dovreste e non dovreste dire. Grazie anche a Pat Mochel, Hanna Linder, Randy Dunlap, Kay Sievers, Vojtech Pavlik, Jan Kara, Josh Boyer, Kees Cook, Andrew Morton, Andi Kleen, Vadim Lobanov, Jesper Juhl, Adrian Bunk, Keri Harris, Frans Pop, David A. Wheeler, Junio Hamano, Michael Kerrisk, e Alex Shepard per le loro revisioni, commenti e contributi. Senza il loro aiuto, questo documento non sarebbe stato possibile.

Manutentore: Greg Kroah-Hartman <[ggreg@kroah.com](mailto:ggreg@kroah.com)>

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/code-of-conduct.rst

### Accordo dei contributori sul codice di condotta

**Warning:** TODO ancora da tradurre

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/development-process.rst

**Translator** Alessia Mantegazza <[amantegazza@vaga.pv.it](mailto:amantegazza@vaga.pv.it)>

### Una guida al processo di sviluppo del Kernel

Contenuti:

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/1.Intro.rst

**Translator** Alessia Mantegazza <[amantegazza@vaga.pv.it](mailto:amantegazza@vaga.pv.it)>

### Introduzione

#### Riepilogo generale

Il resto di questa sezione riguarda il processo di sviluppo del kernel e quella sorta di frustrazione che gli sviluppatori e i loro datori di lavoro potrebbero dover affrontare. Ci sono molte ragioni per le quali del codice per il kernel debba essere incorporato nel kernel ufficiale, fra le quali: disponibilità immediata agli utilizzatori, supporto della comunità in differenti modalità, e la capacità di influenzare la direzione dello sviluppo del kernel. Il codice che contribuisce al kernel Linux deve essere reso disponibile sotto una licenza GPL-compatibile.

La sezione [\*Come funziona il processo di sviluppo\*](#) introduce il processo di sviluppo, il ciclo di rilascio del kernel, ed i meccanismi della finestra d'incorporazione. Il capitolo copre le varie fasi di una modifica: sviluppo, revisione e ciclo d'incorporazione. Ci sono alcuni dibattiti su strumenti e liste di discussione. Gli sviluppatori che sono in attesa di poter sviluppare qualcosa per il kernel sono invitati ad individuare e sistemare bachi come esercizio iniziale.

La sezione [\*I primi passi della pianificazione\*](#) copre i primi stadi della pianificazione di un progetto di sviluppo, con particolare enfasi sul coinvolgimento della comunità, il prima possibile.

La sezione [\*Scrivere codice corretto\*](#) riguarda il processo di scrittura del codice. Qui, sono esposte le diverse insidie che sono state già affrontate da altri sviluppatori. Il capitolo copre

anche alcuni dei requisiti per le modifiche, ed esiste un'introduzione ad alcuni strumenti che possono aiutarvi nell'assicurarvi che le modifiche per il kernel siano corrette.

La sezione [Pubbicare modifiche](#) parla del processo di pubblicazione delle modifiche per la revisione. Per essere prese in considerazione dalla comunità di sviluppo, le modifiche devono essere propriamente formattate ed esposte, e devono essere inviate nel posto giusto. Seguire i consigli presenti in questa sezione dovrebbe essere d'aiuto nell'assicurare la migliore accoglienza possibile del vostro lavoro.

La sezione [Completamento](#) copre ciò che accade dopo la pubblicazione delle modifiche; a questo punto il lavoro è lontano dall'essere concluso. Lavorare con i revisori è una parte cruciale del processo di sviluppo; questa sezione offre una serie di consigli su come evitare problemi in questa importante fase. Gli sviluppatori sono diffidenti nell'affermare che il lavoro è concluso quando una modifica è incorporata nei sorgenti principali.

La sezione [Argomenti avanzati](#) introduce un paio di argomenti “avanzati”: gestire le modifiche con git e controllare le modifiche pubblicate da altri.

La sezione [Per maggiori informazioni](#) chiude il documento con dei riferimenti ad altre fonti che forniscono ulteriori informazioni sullo sviluppo del kernel.

## Di cosa parla questo documento

Il kernel Linux, ha oltre 8 milioni di linee di codice e ben oltre 1000 contributori ad ogni rilascio; è uno dei più vasti e più attivi software liberi progettati mai esistiti. Sin dal suo modesto inizio nel 1991, questo kernel si è evoluto nel miglior componente per sistemi operativi che fanno funzionare piccoli riproduttori musicali, PC, grandi super computer e tutte le altre tipologie di sistemi fra questi estremi. È una soluzione robusta, efficiente ed adattabile a praticamente qualsiasi situazione.

Con la crescita di Linux è arrivato anche un aumento di sviluppatori (ed aziende) desiderosi di partecipare a questo sviluppo. I produttori di hardware vogliono assicurarsi che il loro prodotti siano supportati da Linux, rendendo questi prodotti attrattivi agli utenti Linux. I produttori di sistemi integrati, che usano Linux come componente di un prodotto integrato, vogliono che Linux sia capace ed adeguato agli obiettivi ed il più possibile alla mano. Fornitori ed altri produttori di software che basano i propri prodotti su Linux hanno un chiaro interesse verso capacità, prestazioni ed affidabilità del kernel Linux. E gli utenti finali, anche, spesso vorrebbero cambiare Linux per renderlo più aderente alle proprie necessità.

Una delle caratteristiche più coinvolgenti di Linux è quella dell'accessibilità per gli sviluppatori; chiunque con le capacità richieste può migliorare Linux ed influenzarne la direzione di sviluppo. Prodotti non open-source non possono offrire questo tipo di apertura, che è una caratteristica del software libero. Ma, anzi, il kernel è persino più aperto rispetto a molti altri progetti di software libero. Un classico ciclo di sviluppo trimestrale può coinvolgere 1000 sviluppatori che lavorano per più di 100 differenti aziende (o per nessuna azienda).

Lavorare con la comunità di sviluppo del kernel non è particolarmente difficile. Ma, ciononostante, diversi potenziali contributori hanno trovato delle difficoltà quando hanno cercato di lavorare sul kernel. La comunità del kernel utilizza un proprio modo di operare che gli permette di funzionare agevolmente (e genera un prodotto di alta qualità) in un ambiente dove migliaia di stringhe di codice sono modificate ogni giorno. Quindi non deve sorprendere che il processo di sviluppo del kernel differisca notevolmente dai metodi di sviluppo privati.

Il processo di sviluppo del Kernel può, dall'altro lato, risultare intimidatorio e strano ai nuovi

sviluppatori, ma ha dietro di sé buone ragioni e solide esperienze. Uno sviluppatore che non comprende i modi della comunità del kernel (o, peggio, che cerchi di aggirarli o violarli) avrà un'esperienza deludente nel proprio bagaglio. La comunità di sviluppo, sebbene sia utile a coloro che cercano di imparare, ha poco tempo da dedicare a coloro che non ascoltano o coloro che non sono interessati al processo di sviluppo.

Si spera che coloro che leggono questo documento saranno in grado di evitare queste esperienze spiacevoli. C'è molto materiale qui, ma lo sforzo della lettura sarà ripagato in breve tempo. La comunità di sviluppo ha sempre bisogno di sviluppatori che vogliano aiutare a rendere il kernel migliore; il testo seguente potrebbe esservi d'aiuto - o essere d'aiuto ai vostri collaboratori- per entrare a far parte della nostra comunità.

### Crediti

Questo documento è stato scritto da Jonathan Corbet, [corbet@lwn.net](mailto:corbet@lwn.net). È stato migliorato da Johannes Berg, James Berry, Alex Chiang, Roland Dreier, Randy Dunlap, Jake Edge, Jiri Kosina, Matt Mackall, Arthur Marsh, Amanda McPherson, Andrew Morton, Andrew Price, Tsugikazu Shibata e Jochen Voß.

Questo lavoro è stato supportato dalla Linux Foundation; un ringraziamento speciale ad Amanda McPherson, che ha visto il valore di questo lavoro e lo ha reso possibile.

### L'importanza d'avere il codice nei sorgenti principali

Alcune aziende e sviluppatori ogni tanto si domandano perché dovrebbero preoccuparsi di apprendere come lavorare con la comunità del kernel e di inserire il loro codice nel ramo di sviluppo principale (per ramo principale s'intende quello mantenuto da Linus Torvalds e usato come base dai distributori Linux). Nel breve termine, contribuire al codice può sembrare un costo inutile; può sembra più facile tenere separato il proprio codice e supportare direttamente i suoi utilizzatori. La verità è che il tenere il codice separato ("fuori dai sorgenti", "*out-of-tree*") è un falso risparmio.

Per dimostrare i costi di un codice "fuori dai sorgenti", eccovi alcuni aspetti rilevanti del processo di sviluppo kernel; la maggior parte di essi saranno approfonditi dettagliatamente più avanti in questo documento. Considerate:

- Il codice che è stato inserito nel ramo principale del kernel è disponibile a tutti gli utilizzatori Linux. Sarà automaticamente presente in tutte le distribuzioni che lo consentono. Non c'è bisogno di: driver per dischi, scaricare file, o della scocciatura del dover supportare diverse versioni di diverse distribuzioni; funziona già tutto, per gli sviluppatori e per gli utilizzatori. L'inserimento nel ramo principale risolve un gran numero di problemi di distribuzione e di supporto.
- Nonostante gli sviluppatori kernel si sforzino di tenere stabile l'interfaccia dello spazio utente, quella interna al kernel è in continuo cambiamento. La mancanza di un'interfaccia interna è deliberatamente una decisione di progettazione; ciò permette che i miglioramenti fondamentali vengano fatti in un qualsiasi momento e che risultino fatti con un codice di alta qualità. Ma una delle conseguenze di questa politica è che qualsiasi codice "fuori dai sorgenti" richiede costante manutenzione per renderlo funzionante coi kernel più recenti. Tenere un codice "fuori dai sorgenti" richiede una mole di lavoro significativa solo per farlo funzionare.

Invece, il codice che si trova nel ramo principale non necessita di questo tipo di lavoro poiché ad ogni sviluppatore che faccia una modifica alle interfacce viene richiesto di sistemare anche il codice che utilizza quell'interfaccia. Quindi, il codice che è stato inserito nel ramo principale ha dei costi di mantenimento significativamente più bassi.

- Oltre a ciò, spesso il codice che è all'interno del kernel sarà migliorato da altri sviluppatori. Dare pieni poteri alla vostra comunità di utenti e ai clienti può portare a sorprendenti risultati che migliorano i vostri prodotti.
- Il codice kernel è soggetto a revisioni, sia prima che dopo l'inserimento nel ramo principale. Non importa quanto forti fossero le abilità dello sviluppatore originale, il processo di revisione troverà il modo di migliore il codice. Spesso la revisione trova bachi importanti e problemi di sicurezza. Questo è particolarmente vero per il codice che è stato sviluppato in un ambiente chiuso; tale codice ottiene un forte beneficio dalle revisioni provenienti da sviluppatori esteri. Il codice "fuori dai sorgenti", invece, è un codice di bassa qualità.
- La partecipazione al processo di sviluppo costituisce la vostra via per influenzare la direzione di sviluppo del kernel. Gli utilizzatori che "reclamano da bordo campo" sono ascoltati, ma gli sviluppatori attivi hanno una voce più forte - e la capacità di implementare modifiche che renderanno il kernel più funzionale alle loro necessità.
- Quando il codice è gestito separatamente, esiste sempre la possibilità che terze parti contribuiscano con una differente implementazione che fornisce le stesse funzionalità. Se dovesse accadere, l'inserimento del codice diventerà molto più difficile - fino all'impossibilità. Poi, dovete far fronte a delle alternative poco piacevoli, come: (1) mantenere un elemento non standard "fuori dai sorgenti" per un tempo indefinito, o (2) abbandonare il codice e far migrare i vostri utenti alla versione "nei sorgenti".
- Contribuire al codice è l'azione fondamentale che fa funzionare tutto il processo. Contribuendo attraverso il vostro codice potete aggiungere nuove funzioni al kernel e fornire competenze ed esempi che saranno utili ad altri sviluppatori. Se avete sviluppato del codice Linux (o state pensando di farlo), avete chiaramente interesse nel far proseguire il successo di questa piattaforma. Contribuire al codice è una delle migliori vie per aiutarne il successo.

Il ragionamento sopra citato si applica ad ogni codice "fuori dai sorgenti" dal kernel, incluso il codice proprietario distribuito solamente in formato binario. Ci sono, comunque, dei fattori aggiuntivi che dovrebbero essere tenuti in conto prima di prendere in considerazione qualsiasi tipo di distribuzione binaria di codice kernel. Questo include che:

- Le questioni legali legate alla distribuzione di moduli kernel proprietari sono molto nebbie; parecchi detentori di copyright sul kernel credono che molti moduli binari siano prodotti derivati del kernel e che, come risultato, la loro diffusione sia una violazione della licenza generale di GNU (della quale si parlerà più avanti). L'autore qui non è un avvocato, e niente in questo documento può essere considerato come un consiglio legale. Il vero stato legale dei moduli proprietari può essere determinato esclusivamente da un giudice. Ma l'incertezza che perseguita quei moduli è lì comunque.
- I moduli binari aumentano di molto la difficoltà di fare debugging del kernel, al punto che la maggior parte degli sviluppatori del kernel non vorranno nemmeno tentare. Quindi la diffusione di moduli esclusivamente binari renderà difficile ai vostri utilizzatori trovare un supporto dalla comunità.
- Il supporto è anche difficile per i distributori di moduli binari che devono fornire una versione del modulo per ogni distribuzione e per ogni versione del kernel che vogliono supportare. Per fornire una copertura ragionevole e comprensiva, può essere richiesto di

produrre dozzine di singoli moduli. E inoltre i vostri utilizzatori dovranno aggiornare il vostro modulo separatamente ogni volta che aggiornano il loro kernel.

- Tutto ciò che è stato detto prima riguardo alla revisione del codice si applica doppiamente al codice proprietario. Dato che questo codice non è del tutto disponibile, non può essere revisionato dalla comunità e avrà, senza dubbio, seri problemi.

I produttori di sistemi integrati, in particolare, potrebbero esser tentati dall'evitare molto di ciò che è stato detto in questa sezione, credendo che stiano distribuendo un prodotto finito che utilizza una versione del kernel immutabile e che non richiede un ulteriore sviluppo dopo il rilascio. Questa idea non comprende il valore di una vasta revisione del codice e il valore del permettere ai propri utenti di aggiungere funzionalità al vostro prodotto. Ma anche questi prodotti, hanno una vita commerciale limitata, dopo la quale deve essere rilasciata una nuova versione. A quel punto, i produttori il cui codice è nel ramo principale di sviluppo avranno un codice ben mantenuto e saranno in una posizione migliore per ottenere velocemente un nuovo prodotto pronto per essere distribuito.

## Licenza

IL codice Linux utilizza diverse licenze, ma il codice completo deve essere compatibile con la seconda versione della licenza GNU General Public License (GPLv2), che è la licenza che copre la distribuzione del kernel. Nella pratica, ciò significa che tutti i contributi al codice sono coperti anche'essi dalla GPLv2 (con, opzionalmente, una dicitura che permette la possibilità di distribuirlo con licenze più recenti di GPL) o dalla licenza three-clause BSD. Qualsiasi contributo che non è coperto da una licenza compatibile non verrà accettata nel kernel.

Per il codice sottomesso al kernel non è necessario (o richiesto) la concessione del Copyright. Tutto il codice inserito nel ramo principale del kernel conserva la sua proprietà originale; ne risulta che ora il kernel abbia migliaia di proprietari.

Una conseguenza di questa organizzazione della proprietà è che qualsiasi tentativo di modifica della licenza del kernel è destinata ad un quasi sicuro fallimento. Esistono alcuni scenari pratici nei quali il consenso di tutti i detentori di copyright può essere ottenuto (o il loro codice verrà rimosso dal kernel). Quindi, in sostanza, non esiste la possibilità che si giunga ad una versione 3 della licenza GPL nel prossimo futuro.

È imperativo che tutto il codice che contribuisce al kernel sia legittimamente software libero. Per questa ragione, un codice proveniente da un contributore anonimo (o sotto pseudonimo) non verrà accettato. È richiesto a tutti i contributori di firmare il proprio codice, attestando così che quest'ultimo può essere distribuito insieme al kernel sotto la licenza GPL. Il codice che non è stato licenziato come software libero dal proprio creatore, o che potrebbe creare problemi di copyright per il kernel (come il codice derivante da processi di ingegneria inversa senza le opportune tutele), non può essere diffuso.

Domande relative a questioni legate al copyright sono frequenti nelle liste di discussione dedicate allo sviluppo di Linux. Tali quesiti, normalmente, non riceveranno alcuna risposta, ma una cosa deve essere tenuta presente: le persone che risponderanno a quelle domande non sono avvocati e non possono fornire supporti legali. Se avete questioni legali relative ai sorgenti del codice Linux, non esiste alternativa che quella di parlare con un avvocato esperto nel settore. Fare affidamento sulle risposte ottenute da una lista di discussione tecnica è rischioso.

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/2.Process.rst

**Translator** Alessia Mantegazza <[amanategazza@vaga.pv.it](mailto:amanategazza@vaga.pv.it)>

## Come funziona il processo di sviluppo

Lo sviluppo del Kernel agli inizi degli anni '90 era abbastanza libero, con un numero di utenti e sviluppatori relativamente basso. Con una base di milioni di utenti e con 2000 sviluppatori coinvolti nel giro di un anno, il kernel da allora ha messo in atto un certo numero di procedure per rendere lo sviluppo più agevole. È richiesta una solida conoscenza di come tale processo si svolge per poter esserne parte attiva.

## Il quadro d'insieme

Gli sviluppatori kernel utilizzano un calendario di rilascio generico, dove ogni due o tre mesi viene effettuata un rilascio importante del kernel. I rilasci più recenti sono stati:

5.0	3 marzo, 2019
5.1	5 maggio, 2019
5.2	7 luglio, 2019
5.3	15 settembre, 2019
5.4	24 novembre, 2019
5.5	6 gennaio, 2020

Ciascun rilascio 5.x è un importante rilascio del kernel con nuove funzionalità, modifiche interne dell'API, e molto altro. Un tipico rilascio contiene quasi 13,000 gruppi di modifiche con ulteriori modifiche a parecchie migliaia di linee di codice. La 5.x. è pertanto la linea di confine nello sviluppo del kernel Linux; il kernel utilizza un sistema di sviluppo continuo che integra costantemente nuove importanti modifiche.

Viene seguita una disciplina abbastanza lineare per l'inclusione delle patch di ogni rilascio. All'inizio di ogni ciclo di sviluppo, la "finestra di inclusione" viene dichiarata aperta. In quel momento il codice ritenuto sufficientemente stabile (e che è accettato dalla comunità di sviluppo) viene incluso nel ramo principale del kernel. La maggior parte delle patch per un nuovo ciclo di sviluppo (e tutte le più importanti modifiche) saranno inserite durante questo periodo, ad un ritmo che si attesta sulle 1000 modifiche ("patch" o "gruppo di modifiche") al giorno.

(per inciso, vale la pena notare che i cambiamenti integrati durante la "finestra di inclusione" non escono dal nulla; questi infatti, sono stati raccolti e, verificati in anticipo. Il funzionamento di tale procedimento verrà descritto dettagliatamente più avanti).

La finestra di inclusione resta attiva approssimativamente per due settimane. Al termine di questo periodo, Linus Torvald dichiarerà che la finestra è chiusa e rilascerà il primo degli "rc" del kernel. Per il kernel che è destinato ad essere 5.6, per esempio, il rilascio che emerge al termine della finestra d'inclusione si chiamerà 5.6-rc1. Questo rilascio indica che il momento

di aggiungere nuovi componenti è passato, e che è iniziato il periodo di stabilizzazione del prossimo kernel.

Nelle successive sei/dieci settimane, potranno essere sottoposte solo modifiche che vanno a risolvere delle problematiche. Occasionalmente potrà essere consentita una modifica più consistente, ma tali occasioni sono rare. Gli sviluppatori che tenteranno di aggiungere nuovi elementi al di fuori della finestra di inclusione, tendenzialmente, riceveranno un'accoglienza poco amichevole. Come regola generale: se vi perdete la finestra di inclusione per un dato componente, la cosa migliore da fare è aspettare il ciclo di sviluppo successivo (un'eccezione può essere fatta per i driver per hardware non supportati in precedenza; se toccano codice non facente parte di quello attuale, che non causino regressioni e che potrebbero essere aggiunti in sicurezza in un qualsiasi momento)

Mentre le correzioni si aprono la loro strada all'interno del ramo principale, il ritmo delle modifiche rallenta col tempo. Linus rilascia un nuovo kernel -rc circa una volta alla settimana; e ne usciranno circa 6 o 9 prima che il kernel venga considerato sufficientemente stabile e che il rilascio finale venga fatto. A quel punto tutto il processo ricomincerà.

Esempio: ecco com'è andato il ciclo di sviluppo della versione 5.4 (tutte le date si collocano nel 2018)

15 settembre	5.3 rilascio stabile
30 settembre	5.4-rc1, finestra di inclusione chiusa
6 ottobre	5.4-rc2
13 ottobre	5.4-rc3
20 ottobre	5.4-rc4
27 ottobre	5.4-rc5
3 novembre	5.4-rc6
10 novembre	5.4-rc7
17 novembre	5.4-rc8
24 novembre	5.4 rilascio stabile

In che modo gli sviluppatori decidono quando chiudere il ciclo di sviluppo e creare quindi una rilascio stabile? Un metro valido è il numero di regressioni rilevate nel precedente rilascio. Nessun baco è il benvenuto, ma quelli che procurano problemi su sistemi che hanno funzionato in passato sono considerati particolarmente seri. Per questa ragione, le modifiche che portano ad una regressione sono viste sfavorevolmente e verranno quasi sicuramente annullate durante il periodo di stabilizzazione.

L'obiettivo degli sviluppatori è quello di aggiustare tutte le regressioni conosciute prima che avvenga il rilascio stabile. Nel mondo reale, questo tipo di perfezione difficilmente viene raggiunta; esistono troppe variabili in un progetto di questa portata. Arriva un punto dove ritardare il rilascio finale peggiora la situazione; la quantità di modifiche in attesa della prossima finestra di inclusione crescerà enormemente, creando ancor più regressioni al giro successivo. Quindi molti kernel 5.x escono con una manciata di regressioni delle quali, si spera, nessuna è grave.

Una volta che un rilascio stabile è fatto, il suo costante mantenimento è affidato al "squadra stabilità", attualmente composta da Greg Kroah-Hartman. Questa squadra rilascia occasionalmente degli aggiornamenti relativi al rilascio stabile usando la numerazione 5.x.y. Per essere presa in considerazione per un rilascio d'aggiornamento, una modifica deve: (1) correggere un baco importante (2) essere già inserita nel ramo principale per il prossimo sviluppo del kernel. Solitamente, passato il loro rilascio iniziale, i kernel ricevono aggiornamenti per più di un ciclo di sviluppo. Quindi, per esempio, la storia del kernel 5.2 appare così (anno 2019):

7 luglio	5.2 rilascio stabile
14 luglio	5.2.1
21 luglio	5.2.2
26 luglio	5.2.3
28 luglio	5.2.4
31 luglio	5.2.5
...	...
11 ottobre	5.2.21

La 5.2.21 fu l'aggiornamento finale per la versione 5.2.

Alcuni kernel sono destinati ad essere kernel a “lungo termine”; questi riceveranno assistenza per un lungo periodo di tempo. Al momento in cui scriviamo, i manutentori dei kernel stabili a lungo termine sono:

3.16	Ben Hutchings	(kernel stabile molto più a lungo termine)
4.4	Greg Kroah-Hartman e Sasha Levin	(kernel stabile molto più a lungo termine)
4.9	Greg Kroah-Hartman e Sasha Levin	
4.14	Greg Kroah-Hartman e Sasha Levin	
4.19	Greg Kroah-Hartman e Sasha Levin	
5.4i	Greg Kroah-Hartman e Sasha Levin	

Questa selezione di kernel di lungo periodo sono puramente dovuti ai loro manutentori, alla loro necessità e al tempo per tenere aggiornate proprio quelle versioni. Non ci sono altri kernel a lungo termine in programma per alcun rilascio in arrivo.

## Il ciclo di vita di una patch

Le patch non passano direttamente dalla tastiera dello sviluppatori al ramo principale del kernel. Esiste, invece, una procedura disegnata per assicurare che ogni patch sia di buona qualità e desiderata nel ramo principale. Questo processo avviene velocemente per le correzioni meno importanti, o, nel caso di patch ampie e controverse, va avanti per anni. Per uno sviluppatore la maggior frustrazione viene dalla mancanza di comprensione di questo processo o dai tentativi di aggirarlo.

Nella speranza di ridurre questa frustrazione, questo documento spiegherà come una patch viene inserita nel kernel. Ciò che segue è un'introduzione che descrive il processo ideale. Approfondimenti verranno invece trattati più avanti.

Una patch attraversa, generalmente, le seguenti fasi:

- Progetto. In questa fase sono stabilite quelli che sono i requisiti della modifica - e come verranno soddisfatti. Il lavoro di progettazione viene spesso svolto senza coinvolgere la

comunità, ma è meglio renderlo il più aperto possibile; questo può far risparmiare molto tempo evitando eventuali riprogettazioni successive.

- Prima revisione. Le patch vengono pubblicate sulle liste di discussione interessate, e gli sviluppatori in quella lista risponderanno coi loro commenti. Se si svolge correttamente, questo procedimento potrebbe far emergere problemi rilevanti in una patch.
- Revisione più ampia. Quando la patch è quasi pronta per essere inserita nel ramo principale, un manutentore importante del sottosistema dovrebbe accettarla - anche se, questa accettazione non è una garanzia che la patch arriverà nel ramo principale. La patch sarà visibile nei sorgenti del sottosistema in questione e nei sorgenti -next (descritti sotto). Quando il processo va a buon fine, questo passo porta ad una revisione più estesa della patch e alla scoperta di problemi d'integrazione con il lavoro altrui.
- Per favore, tenete da conto che la maggior parte dei manutentori ha anche un lavoro quotidiano, quindi integrare le vostre patch potrebbe non essere la loro priorità più alta. Se una vostra patch riceve dei suggerimenti su dei cambiamenti necessari, dovreste applicare quei cambiamenti o giustificare perché non sono necessari. Se la vostra patch non riceve alcuna critica ma non è stata integrata dal manutentore del driver o sottosistema, allora dovreste continuare con i necessari aggiornamenti per mantenere la patch aggiornata al kernel più recente cosicché questa possa integrarsi senza problemi; continuate ad inviare gli aggiornamenti per essere revisionati e integrati.
- Inclusione nel ramo principale. Eventualmente, una buona patch verrà inserita all'interno nel repository principale, gestito da Linus Torvalds. In questa fase potrebbero emergere nuovi problemi e/o commenti; è importante che lo sviluppatore sia collaborativo e che sistemi ogni questione che possa emergere.
- Rilascio stabile. Ora, il numero di utilizzatori che sono potenzialmente toccati dalla patch è aumentato, quindi, ancora una volta, potrebbero emergere nuovi problemi.
- Manutenzione di lungo periodo. Nonostante sia possibile che uno sviluppatore si dimentichi del codice dopo la sua integrazione, questo comportamento lascia una brutta impressione nella comunità di sviluppo. Integrare il codice elimina alcuni degli oneri facenti parte della manutenzione, in particolare, sistemerà le problematiche causate dalle modifiche all'API. Ma lo sviluppatore originario dovrebbe continuare ad assumersi la responsabilità per il codice se quest'ultimo continua ad essere utile nel lungo periodo.

Uno dei più grandi errori fatti dagli sviluppatori kernel (o dai loro datori di lavoro) è quello di cercare di ridurre tutta la procedura ad una singola "integrazione nel ramo principale". Questo approccio inevitabilmente conduce a una condizione di frustrazione per tutti coloro che sono coinvolti.

### Come le modifiche finiscono nel Kernel

Esiste una sola persona che può inserire le patch nel repository principale del kernel: Linus Torvalds. Ma, per esempio, di tutte le 9500 patch che entrarono nella versione 2.6.38 del kernel, solo 112 (circa l'1,3%) furono scelte direttamente da Linus in persona. Il progetto del kernel è cresciuto fino a raggiungere una dimensione tale per cui un singolo sviluppatore non può controllare e selezionare indipendentemente ogni modifica senza essere supportato. La via scelta dagli sviluppatori per indirizzare tale crescita è stata quella di utilizzare un sistema di "sottotenenti" basato sulla fiducia.

Il codice base del kernel è spezzato in una serie di sottosistemi: rete, supporto per specifiche

architetture, gestione della memoria, video e strumenti, etc. Molti sottosistemi hanno un manutentore designato: ovvero uno sviluppatore che ha piena responsabilità di tutto il codice presente in quel sottosistema. Tali manutentori di sottosistema sono i guardiani (in un certo senso) della parte di kernel che gestiscono; sono coloro che (solitamente) accetteranno una patch per l'inclusione nel ramo principale del kernel.

I manutentori di sottosistema gestiscono ciascuno la propria parte dei sorgenti del kernel, utilizzando abitualmente (ma certamente non sempre) git. Strumenti come git (e affini come quilt o mercurial) permettono ai manutentori di stilare una lista delle patch, includendo informazioni sull'autore ed altri metadati. In ogni momento, il manutentore può individuare quale patch nel suo repository non si trova nel ramo principale.

Quando la "finestra di integrazione" si apre, i manutentori di alto livello chiederanno a Linus di "prendere" dai loro repository le modifiche che hanno selezionato per l'inclusione. Se Linus acconsente, il flusso di patch si convoglierà nel repository di quest'ultimo, divenendo così parte del ramo principale del kernel. La quantità d'attenzione che Linus presta alle singole patch ricevute durante l'operazione di integrazione varia. È chiaro che, qualche volta, guardi più attentamente. Ma, come regola generale, Linus confida nel fatto che i manutentori di sottosistema non selezionino pessime patch.

I manutentori di sottosistemi, a turno, possono "prendere" patch provenienti da altri manutentori. Per esempio, i sorgenti per la rete sono costruiti da modifiche che si sono accumulate inizialmente nei sorgenti dedicati ai driver per dispositivi di rete, rete senza fili, ecc. Tale catena di repository può essere più o meno lunga, benché raramente ecceda i due o tre collegamenti. Questo processo è conosciuto come "la catena della fiducia", perché ogni manutentore all'interno della catena si fida di coloro che gestiscono i livelli più bassi.

Chiaramente, in un sistema come questo, l'inserimento delle patch all'interno del kernel si basa sul trovare il manutentore giusto. Di norma, inviare patch direttamente a Linus non è la via giusta.

## Sorgenti -next

La catena di sottosistemi guida il flusso di patch all'interno del kernel, ma solleva anche un interessante quesito: se qualcuno volesse vedere tutte le patch pronte per la prossima finestra di integrazione? Gli sviluppatori si interesseranno alle patch in sospeso per verificare che non ci siano altri conflitti di cui preoccuparsi; una modifica che, per esempio, cambia il prototipo di una funzione fondamentale del kernel andrà in conflitto con qualsiasi altra modifica che utilizzi la vecchia versione di quella funzione. Revisori e tester vogliono invece avere accesso alle modifiche nella loro totalità prima che approdino nel ramo principale del kernel. Uno potrebbe prendere le patch provenienti da tutti i sottosistemi d'interesse, ma questo sarebbe un lavoro enorme e fallace.

La risposta ci viene sotto forma di sorgenti -next, dove i sottosistemi sono raccolti per essere testati e controllati. Il più vecchio di questi sorgenti, gestito da Andrew Morton, è chiamato "-mm" (memory management, che è l'inizio di tutto). L-mm integra patch proveniente da una lunga lista di sottosistemi; e ha, inoltre, alcune patch destinate al supporto del debugging.

Oltre a questo, -mm contiene una raccolta significativa di patch che sono state selezionate da Andrew direttamente. Queste patch potrebbero essere state inviate in una lista di discussione, o possono essere applicate ad una parte del kernel per la quale non esiste un sottosistema dedicato. Di conseguenza, -mm opera come una specie di sottosistema "ultima spiaggia"; se per una patch non esiste una via chiara per entrare nel ramo principale, allora è probabile

che finirà in -mm. Le patch passate per -mm eventualmente finiranno nel sottosistema più appropriato o saranno inviate direttamente a Linus. In un tipico ciclo di sviluppo, circa il 5-10% delle patch andrà nel ramo principale attraverso -mm.

La patch -mm correnti sono disponibili nella cartella "mmotm" (-mm of the moment) all'indirizzo:

<http://www.ozlabs.org/~akpm/mmotm/>

È molto probabile che l'uso dei sorgenti MMOTM diventi un'esperienza frustrante; ci sono buone probabilità che non compili nemmeno.

I sorgenti principali per il prossimo ciclo d'integrazione delle patch è linux-next, gestito da Stephen Rothwell. I sorgenti linux-next sono, per definizione, un'istantanea di come dovrà apparire il ramo principale dopo che la prossima finestra di inclusione si chiuderà. I linux-next sono annunciati sulla lista di discussione linux-kernel e linux-next nel momento in cui vengono assemblati; e possono essere scaricate da:

<http://www.kernel.org/pub/linux/kernel/next/>

Linux-next è divenuto parte integrante del processo di sviluppo del kernel; tutte le patch incorporate durante una finestra di integrazione dovrebbero aver trovato la propria strada in linux-next, a volte anche prima dell'apertura della finestra di integrazione.

### Sorgenti in preparazione

Nei sorgenti del kernel esiste la cartella drivers/staging/, dove risiedono molte sotto-cartelle per i driver o i filesystem che stanno per essere aggiunti al kernel. Questi restano nella cartella drivers/staging fintanto che avranno bisogno di maggior lavoro; una volta completato, possono essere spostate all'interno del kernel nel posto più appropriato. Questo è il modo di tener traccia dei driver che non sono ancora in linea con gli standard di codifica o qualità, ma che le persone potrebbero voler usare ugualmente e tracciarne lo sviluppo.

Greg Kroah-Hartman attualmente gestisce i sorgenti in preparazione. I driver che non sono completamente pronti vengono inviati a lui, e ciascun driver avrà la propria sotto-cartella in drivers/staging/. Assieme ai file sorgenti dei driver, dovrebbe essere presente nella stessa cartella anche un file TODO. Il file TODO elenca il lavoro ancora da fare su questi driver per poter essere accettati nel kernel, e indica anche la lista di persone da inserire in copia conoscenza per ogni modifica fatta. Le regole attuali richiedono che i driver debbano, come minimo, compilare adeguatamente.

La *preparazione* può essere una via relativamente facile per inserire nuovi driver all'interno del ramo principale, dove, con un po' di fortuna, saranno notati da altri sviluppatori e migliorati velocemente. Entrare nella fase di preparazione non è però la fine della storia, infatti, il codice che si trova nella cartella staging che non mostra regolari progressi potrebbe essere rimosso. Le distribuzioni, inoltre, tendono a dimostrarsi relativamente riluttanti nell'attivare driver in preparazione. Quindi lo preparazione è, nel migliore dei casi, una tappa sulla strada verso il divenire un driver del ramo principale.

## Strumenti

Come è possibile notare dal testo sopra, il processo di sviluppo del kernel dipende pesantemente dalla capacità di guidare la raccolta di patch in diverse direzioni. L'intera cosa non funzionerebbe se non venisse svolta con l'uso di strumenti appropriati e potenti. Spiegare l'uso di tali strumenti non è lo scopo di questo documento, ma c'è spazio per alcuni consigli.

In assoluto, nella comunità del kernel, predomina l'uso di git come sistema di gestione dei sorgenti. Git è una delle diverse tipologie di sistemi distribuiti di controllo versione che sono stati sviluppati nella comunità del software libero. Esso è calibrato per lo sviluppo del kernel, e si comporta abbastanza bene quando ha a che fare con repository grandi e con un vasto numero di patch. Git ha inoltre la reputazione di essere difficile da imparare e utilizzare, benché stia migliorando. Agli sviluppatori del kernel viene richiesta un po' di familiarità con git; anche se non lo utilizzano per il proprio lavoro, hanno bisogno di git per tenersi al passo con il lavoro degli altri sviluppatori (e con il ramo principale).

Git è ora compreso in quasi tutte le distribuzioni Linux. Esiste un sito che potete consultare:

<http://git-scm.com/>

Qui troverete i riferimenti alla documentazione e alle guide passo-passo.

Tra gli sviluppatori Kernel che non usano git, la scelta alternativa più popolare è quasi sicuramente Mercurial:

<http://www.selenic.com/mercurial/>

Mercurial condivide diverse caratteristiche con git, ma fornisce un'interfaccia che potrebbe risultare più semplice da utilizzare.

L'altro strumento che vale la pena conoscere è Quilt:

<http://savannah.nongnu.org/projects/quilt/>

Quilt è un sistema di gestione delle patch, piuttosto che un sistema di gestione dei sorgenti. Non mantiene uno storico degli eventi; ma piuttosto è orientato verso il tracciamento di uno specifico insieme di modifiche rispetto ad un codice in evoluzione. Molti dei più grandi manutentori di sottosistema utilizzano quilt per gestire le patch che dovrebbero essere integrate. Per la gestione di certe tipologie di sorgenti (-mm, per esempio), quilt è il miglior strumento per svolgere il lavoro.

## Liste di discussione

Una grossa parte del lavoro di sviluppo del Kernel Linux viene svolto tramite le liste di discussione. È difficile essere un membro della comunità pienamente coinvolto se non si partecipa almeno ad una lista da qualche parte. Ma, le liste di discussione di Linux rappresentano un potenziale problema per gli sviluppatori, che rischiano di venir sepolti da un mare di email, restare incagliati nelle convenzioni in vigore nelle liste Linux, o entrambi.

Molte delle liste di discussione del Kernel girano su vger.kernel.org; l'elenco principale lo si trova sul sito:

<http://vger.kernel.org/vger-lists.html>

Esistono liste gestite altrove; un certo numero di queste sono in redhat.com/mailman/listinfo.

La lista di discussione principale per lo sviluppo del kernel è, ovviamente, linux-kernel. Questa lista è un luogo ostile dove trovarsi; i volumi possono raggiungere i 500 messaggi al giorno, la quantità di "rumore" è elevata, la conversazione può essere strettamente tecnica e i partecipanti non sono sempre preoccupati di mostrare un alto livello di educazione. Ma non esiste altro luogo dove la comunità di sviluppo del kernel si unisce per intero; gli sviluppatori che evitano tale lista si perderanno informazioni importanti.

Ci sono alcuni consigli che possono essere utili per sopravvivere a linux-kernel:

- Tenete la lista in una cartella separata, piuttosto che inserirla nella casella di posta principale. Così da essere in grado di ignorare il flusso di mail per un certo periodo di tempo.
- Non cercate di seguire ogni conversazione - nessuno lo fa. È importante filtrare solo gli argomenti d'interesse (sebbene va notato che le conversazioni di lungo periodo possono deviare dall'argomento originario senza cambiare il titolo della mail) e le persone che stanno partecipando.
- Non alimentate i troll. Se qualcuno cerca di creare nervosismo, ignoratelo.
- Quando rispondete ad una mail linux-kernel (o ad altre liste) mantenete tutti i Cc:. In assenza di importanti motivazioni (come una richiesta esplicita), non dovreste mai togliere destinatari. Assicuratevi sempre che la persona alla quale state rispondendo sia presente nella lista Cc. Questa usanza fa sì che divenga inutile chiedere esplicitamente di essere inseriti in copia nel rispondere al vostro messaggio.
- Cercate nell'archivio della lista (e nella rete nella sua totalità) prima di far domande. Molti sviluppatori possono divenire impazienti con le persone che chiaramente non hanno svolto i propri compiti a casa.
- Evitate il *top-posting* (cioè la pratica di mettere la vostra risposta sopra alla frase alla quale state rispondendo). Ciò renderebbe la vostra risposta difficile da leggere e genera scarsa impressione.
- Chiedete nella lista di discussione corretta. Linux-kernel può essere un punto di incontro generale, ma non è il miglior posto dove trovare sviluppatori da tutti i sottosistemi.

Infine, la ricerca della corretta lista di discussione è uno degli errori più comuni per gli sviluppatori principianti. Qualcuno che pone una domanda relativa alla rete su linux-kernel riceverà quasi certamente il suggerimento di chiedere sulla lista netdev, che è la lista frequentata dagli sviluppatori di rete. Ci sono poi altre liste per i sottosistemi SCSI, video4linux, IDE, filesystem, etc. Il miglior posto dove cercare una lista di discussione è il file MAINTAINERS che si trova nei sorgenti del kernel.

## Iniziare con lo sviluppo del Kernel

Sono comuni le domande sul come iniziare con lo sviluppo del kernel - sia da singole persone che da aziende. Altrettanto comuni sono i passi falsi che rendono l'inizio di tale relazione più difficile di quello che dovrebbe essere.

Le aziende spesso cercano di assumere sviluppatori noti per creare un gruppo di sviluppo iniziale. Questo, in effetti, può essere una tecnica efficace. Ma risulta anche essere dispendiosa e non va ad accrescere il bacino di sviluppatori kernel con esperienza. È possibile anche "portare a casa" sviluppatori per accelerare lo sviluppo del kernel, dando comunque all'investimento un po' di tempo. Prendersi questo tempo può fornire al datore di lavoro un gruppo di sviluppatori

che comprendono sia il kernel che l'azienda stessa, e che possono supportare la formazione di altre persone. Nel medio periodo, questa è spesso uno delle soluzioni più proficue.

I singoli sviluppatori sono spesso, comprensibilmente, una perdita come punto di partenza. Iniziare con un grande progetto può rivelarsi intimidatorio; spesso all'inizio si vuole solo verificare il terreno con qualcosa di piccolo. Questa è una delle motivazioni per le quali molti sviluppatori saltano alla creazione di patch che vanno a sistemare errori di battitura o problematiche minori legate allo stile del codice. Sfortunatamente, tali patch creano un certo livello di rumore che distrae l'intera comunità di sviluppo, quindi, sempre di più, esse vengono degradate. I nuovi sviluppatori che desiderano presentarsi alla comunità non riceveranno l'accoglienza che vorrebbero con questi mezzi.

Andrew Morton da questo consiglio agli aspiranti sviluppatori kernel

Il primo progetto per un neofita del kernel dovrebbe essere sicuramente quello di "assicurarsi che il kernel funzioni alla perfezione sempre e su tutte le macchine sulle quali potete stendere la vostra mano". Solitamente il modo per fare ciò è quello di collaborare con gli altri nel sistemare le cose (questo richiede persistenza!) ma va bene - è parte dello sviluppo kernel.

(<http://lwn.net/Articles/283982/>).

In assenza di problemi ovvi da risolvere, si consiglia agli sviluppatori di consultare, in generale, la lista di regressioni e di bachi aperti. Non c'è mai carenza di problematiche bisognose di essere sistematiche; accollandosi tali questioni gli sviluppatori accumuleranno esperienza con la procedura, ed allo stesso tempo, aumenteranno la loro rispettabilità all'interno della comunità di sviluppo.

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/3.Early-stage.rst

**Translator** Alessia Mantegazza <[amantegazza@vaga.pv.it](mailto:amantegazza@vaga.pv.it)>

### I primi passi della pianificazione

Osservando un progetto di sviluppo per il kernel Linux, si potrebbe essere tentati dal saltare tutto e iniziare a codificare. Tuttavia, come ogni progetto significativo, molta della preparazione per giungere al successo viene fatta prima che una sola linea di codice venga scritta. Il tempo speso nella pianificazione e la comunicazione può far risparmiare molto tempo in futuro.

### Specificare il problema

Come qualsiasi progetto ingegneristico, un miglioramento del kernel di successo parte con una chiara descrizione del problema da risolvere. In alcuni casi, questo passaggio è facile: ad esempio quando un driver è richiesto per un particolare dispositivo. In altri casi invece, si tende a confondere il problema reale con le soluzioni proposte e questo può portare all'emergere di problemi.

Facciamo un esempio: qualche anno fa, gli sviluppatori che lavoravano con linux audio cercarono un modo per far girare le applicazioni senza dropouts o altri artefatti dovuti all'eccessivo ritardo nel sistema. La soluzione alla quale giunsero fu un modulo del kernel destinato ad aggiornarsi al framework Linux Security Module (LSM); questo modulo poteva essere configurato per dare ad una specifica applicazione accesso allo schedulatore *realtime*. Tale modulo fu implementato e inviato nella lista di discussione linux-kernel, dove incontrò subito dei problemi.

Per gli sviluppatori audio, questo modulo di sicurezza era sufficiente a risolvere il loro problema nell'immediato. Per l'intera comunità kernel, invece, era un uso improprio del framework LSM (che non è progettato per conferire privilegi a processi che altrimenti non avrebbero potuto ottenerli) e un rischio per la stabilità del sistema. Le loro soluzioni di punta nel breve periodo, comportavano un accesso alla schedulazione realtime attraverso il meccanismo rlimit, e nel lungo periodo un costante lavoro nella riduzione dei ritardi.

La comunità audio, comunque, non poteva vedere al di là della singola soluzione che avevano implementato; erano riluttanti ad accettare alternative. Il conseguente dissenso lasciò in quegli sviluppatori un senso di disillusione nei confronti dell'intero processo di sviluppo; uno di loro scrisse questo messaggio:

Ci sono numerosi sviluppatori del kernel Linux davvero bravi, ma rischiano di restare sovrastati da una vasta massa di stolti arroganti. Cercare di comunicare le richieste degli utenti a queste persone è una perdita di tempo. Loro sono troppo "intelligenti" per stare ad ascoltare dei poveri mortali.

(<http://lwn.net/Articles/131776/>).

La realtà delle cose fu differente; gli sviluppatori del kernel erano molto più preoccupati per la stabilità del sistema, per la manutenzione di lungo periodo e cercavano la giusta soluzione alla problematica esistente con uno specifico modulo. La morale della storia è quella di concentrarsi sul problema - non su di una specifica soluzione- e di discuterne con la comunità di sviluppo prima di investire tempo nella scrittura del codice.

Quindi, osservando un progetto di sviluppo del kernel, si dovrebbe rispondere a questa lista di domande:

- Qual'è, precisamente, il problema che dev'essere risolto?
- Chi sono gli utenti coinvolti da tal problema? A quale caso dovrebbe essere indirizzata la soluzione?

- In che modo il kernel risulta manchevole nell'indirizzare il problema in questione?

Solo dopo ha senso iniziare a considerare le possibili soluzioni.

## Prime discussioni

Quando si pianifica un progetto di sviluppo per il kernel, sarebbe quanto meno opportuno discuterne inizialmente con la comunità prima di lanciarsi nell'implementazione. Una discussione preliminare può far risparmiare sia tempo che problemi in svariati modi:

- Potrebbe essere che il problema sia già stato risolto nel kernel in una maniera che non avete ancora compreso. Il kernel Linux è grande e ha una serie di funzionalità e capacità che non sono scontate nell'immediato. Non tutte le capacità del kernel sono documentate così bene come ci piacerebbe, ed è facile perdersi qualcosa. Il vostro autore ha assistito alla pubblicazione di un driver intero che duplica un altro driver esistente di cui il nuovo autore era ignaro. Il codice che rinnova ingranaggi già esistenti non è soltanto dispendioso; non verrà nemmeno accettato nel ramo principale del kernel.
- Potrebbero esserci proposte che non sono considerate accettabili per l'integrazione all'interno del ramo principale. È meglio affrontarle prima di scrivere il codice.
- È possibile che altri sviluppatori abbiano pensato al problema; potrebbero avere delle idee per soluzioni migliori, e potrebbero voler contribuire alla loro creazione.

Anni di esperienza con la comunità di sviluppo del kernel hanno impartito una chiara lezione: il codice per il kernel che è pensato e sviluppato a porte chiuse, inevitabilmente, ha problematiche che si rivelano solo quando il codice viene rilasciato pubblicamente. Qualche volta tali problemi sono importanti e richiedono mesi o anni di sforzi prima che il codice possa raggiungere gli standard richiesti della comunità. Alcuni esempi possono essere:

- La rete Devicescape è stata creata e implementata per sistemi mono-processore. Non avrebbe potuto essere inserita nel ramo principale fino a che non avesse supportato anche i sistemi multi-processore. Riadattare i meccanismi di sincronizzazione e simili è un compito difficile; come risultato, l'inserimento di questo codice (ora chiamato mac80211) fu rimandato per più di un anno.
- Il filesystem Reiser4 include una serie di funzionalità che, secondo l'opinione degli sviluppatori principali del kernel, avrebbero dovuto essere implementate a livello di filesystem virtuale. Comprende anche funzionalità che non sono facilmente implementabili senza esporre il sistema al rischio di uno stallo. La scoperta tardiva di questi problemi - e il diniego a risolvere alcuni - ha avuto come conseguenza il fatto che Reiser4 resta fuori dal ramo principale del kernel.
- Il modulo di sicurezza AppArmor utilizzava strutture dati del filesystem virtuale interno in modi che sono stati considerati rischiosi e inattendibili. Questi problemi (tra le altre cose) hanno tenuto AppArmor fuori dal ramo principale per anni.

Ciascuno di questi casi è stato un travaglio e ha richiesto del lavoro straordinario, cose che avrebbero potuto essere evitate con alcune "chiacchierate" preliminari con gli sviluppatori kernel.

### Con chi parlare?

Quando gli sviluppatori hanno deciso di rendere pubblici i propri progetti, la domanda successiva sarà: da dove partiamo? La risposta è quella di trovare la giusta lista di discussione e il giusto manutentore. Per le liste di discussione, il miglior approccio è quello di cercare la lista più adatta nel file MAINTAINERS. Se esiste una lista di discussione di sottosistema, è preferibile pubblicare lì piuttosto che sulla lista di discussione generale del kernel Linux; avrete maggiori probabilità di trovare sviluppatori con esperienza sul tema, e l'ambiente che troverete potrebbe essere più incoraggiante.

Trovare manutentori può rivelarsi un po' difficoltoso. Ancora, il file MAINTAINERS è il posto giusto da dove iniziare. Il file potrebbe non essere sempre aggiornato, inoltre, non tutti i sottosistemi sono rappresentati qui. Coloro che sono elencati nel file MAINTAINERS potrebbero, in effetti, non essere le persone che attualmente svolgono quel determinato ruolo. Quindi, quando c'è un dubbio su chi contattare, un trucco utile è quello di usare git (git log in particolare) per vedere chi attualmente è attivo all'interno del sottosistema interessato. Controllate chi sta scrivendo le patch, e chi, se non ci fosse nessuno, sta aggiungendo la propria firma (Signed-off-by) a quelle patch. Quelle sono le persone maggiormente qualificate per aiutarvi con lo sviluppo di nuovo progetto.

Il compito di trovare il giusto manutentore, a volte, è una tale sfida che ha spinto gli sviluppatori del kernel a scrivere uno script che li aiutasse in questa ricerca:

```
.../scripts/get_maintainer.pl
```

Se questo script viene eseguito con l'opzione “-f” ritornerà il manutentore(i) attuale per un dato file o cartella. Se viene passata una patch sulla linea di comando, lo script elencherà i manutentori che dovrebbero riceverne una copia. Ci sono svariate opzioni che regolano quanto a fondo get\_maintainer.pl debba cercare i manutentori; state quindi prudenti nell'utilizzare le opzioni più aggressive poiché potreste finire per includere sviluppatori che non hanno un vero interesse per il codice che state modificando.

Se tutto ciò dovesse fallire, parlare con Andrew Morton potrebbe essere un modo efficace per capire chi è il manutentore di un dato pezzo di codice.

### Quando pubblicare

Se potete, pubblicate i vostri intenti durante le fasi preliminari, sarà molto utile. Descrivete il problema da risolvere e ogni piano che è stato elaborato per l'implementazione. Ogni informazione fornita può aiutare la comunità di sviluppo a fornire spunti utili per il progetto.

Un evento che potrebbe risultare scoraggiate e che potrebbe accadere in questa fase non è il ricevere una risposta ostile, ma, invece, ottenere una misera o inesistente reazione. La triste verità è che: (1) gli sviluppatori del kernel tendono ad essere occupati, (2) ci sono tante persone con grandi progetti e poco codice (o anche solo la prospettiva di avere un codice) a cui riferirsi e (3) nessuno è obbligato a revisionare o a fare osservazioni in merito ad idee pubblicate da altri. Oltre a questo, progetti di alto livello spesso nascondono problematiche che si rivelano solo quando qualcuno cerca di implementarle; per questa ragione gli sviluppatori kernel preferirebbero vedere il codice.

Quindi, se una richiesta pubblica di commenti riscuote poco successo, non pensate che ciò significhi che non ci sia interesse nel progetto. Sfortunatamente, non potete nemmeno assumere

che non ci siano problemi con la vostra idea. La cosa migliore da fare in questa situazione è quella di andare avanti e tenere la comunità informata mentre procedete.

## Ottenere riscontri ufficiali

Se il vostro lavoro è stato svolto in un ambiente aziendale - come molto del lavoro fatto su Linux - dovete, ovviamente, avere il permesso dei dirigenti prima che possiate pubblicare i progetti, o il codice aziendale, su una lista di discussione pubblica. La pubblicazione di codice che non è stato rilascio espressamente con licenza GPL-compatibile può rivelarsi problematico; prima la dirigenza, e il personale legale, troverà una decisione sulla pubblicazione di un progetto, meglio sarà per tutte le persone coinvolte.

A questo punto, alcuni lettori potrebbero pensare che il loro lavoro sul kernel è preposto a supportare un prodotto che non è ancora ufficialmente riconosciuto. Rivelare le intenzioni dei propri datori di lavori in una lista di discussione pubblica potrebbe non essere una soluzione valida. In questi casi, vale la pena considerare se la segretezza sia necessaria o meno; spesso non c'è una reale necessità di mantenere chiusi i progetti di sviluppo.

Detto ciò, ci sono anche casi dove l'azienda legittimamente non può rivelare le proprie intenzioni in anticipo durante il processo di sviluppo. Le aziende che hanno sviluppatori kernel esperti possono scegliere di procedere a carte coperte partendo dall'assunto che saranno in grado di evitare, o gestire, in futuro, eventuali problemi d'integrazione. Per le aziende senza questo tipo di esperti, la migliore opzione è spesso quella di assumere uno sviluppatore esterno che revisioni i progetti con un accordo di segretezza. La Linux Foundation applica un programma di NDA creato appositamente per aiutare le aziende in questa particolare situazione; potrete trovare più informazioni sul sito:

[http://www.linuxfoundation.org/en/NDA\\_program](http://www.linuxfoundation.org/en/NDA_program)

Questa tipologia di revisione è spesso sufficiente per evitare gravi problemi senza che sia richiesta l'esposizione pubblica del progetto.

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/4.Coding.rst

**Translator** Alessia Mantegazza <[amantegazza@vaga.pv.it](mailto:amantegazza@vaga.pv.it)>

## Scrivere codice corretto

Nonostante ci sia molto da dire sul processo di creazione, sulla sua solidità e sul suo orientamento alla comunità, la prova di ogni progetto di sviluppo del kernel si trova nel codice stesso. È il codice che sarà esaminato dagli altri sviluppatori ed inserito (o no) nel ramo principale. Quindi è la qualità di questo codice che determinerà il successo finale del progetto.

Questa sezione esaminerà il processo di codifica. Inizieremo con uno sguardo sulle diverse casistiche nelle quali gli sviluppatori kernel possono sbagliare. Poi, l'attenzione si sposterà verso "il fare le cose correttamente" e sugli strumenti che possono essere utili in questa missione.

### Trappole

#### Lo stile del codice

Il kernel ha da tempo delle norme sullo stile di codifica che sono descritte in [Stile del codice per il kernel Linux](#). Per la maggior parte del tempo, la politica descritta in quel file è stata praticamente informativa. Ne risulta che ci sia una quantità sostanziale di codice nel kernel che non rispetta le linee guida relative allo stile. La presenza di quel codice conduce a due distinti pericoli per gli sviluppatori kernel.

Il primo di questi è credere che gli standard di codifica del kernel non sono importanti e possono non essere applicati. La verità è che aggiungere nuovo codice al kernel è davvero difficile se questo non rispetta le norme; molti sviluppatori richiederanno che il codice sia riformulato prima che anche solo lo revisionino. Una base di codice larga quanto il kernel richiede una certa uniformità, in modo da rendere possibile per gli sviluppatori una comprensione veloce di ogni sua parte. Non ci sono, quindi, più spazi per un codice formattato alla carlona.

Occasionalmente, lo stile di codifica del kernel andrà in conflitto con lo stile richiesto da un datore di lavoro. In alcuni casi, lo stile del kernel dovrà prevalere prima che il codice venga inserito. Mettere il codice all'interno del kernel significa rinunciare a un certo grado di controllo in differenti modi - incluso il controllo sul come formattare il codice.

L'altra trappola è quella di pensare che il codice già presente nel kernel abbia urgentemente bisogno di essere sistemato. Gli sviluppatori potrebbero iniziare a generare patch che correggono lo stile come modo per prendere familiarità con il processo, o come modo per inserire i propri nomi nei changelog del kernel - o entrambe. La comunità di sviluppo vede un'attività di codifica puramente correttiva come "rumore"; queste attività riceveranno una fredda accoglienza. Di conseguenza è meglio evitare questo tipo di patch. Mentre si lavora su un pezzo di codice è normale correggerne anche lo stile, ma le modifiche di stile non dovrebbero essere fatte fini a se stesse.

Il documento sullo stile del codice non dovrebbe essere letto come una legge assoluta che non può mai essere trasgredita. Se c'è un a buona ragione (per esempio, una linea che diviene poco leggibile se divisa per rientrare nel limite di 80 colonne), fatelo e basta.

Notate che potete utilizzare lo strumento "clang-format" per aiutarvi con le regole, per una ri-formattazione automatica e veloce del vostro codice e per revisionare interi file per individuare errori nello stile di codifica, refusi e possibili miglioramenti. Inoltre è utile anche per classificare gli `#includes`, per allineare variabili/macro, per testi derivati ed altri compiti del genere. Consultate il file [clang-format](#) per maggiori dettagli

#### Livelli di astrazione

I professori di Informatica insegnano ai propri studenti a fare ampio uso dei livelli di astrazione nel nome della flessibilità e del nascondere informazioni. Certo il kernel fa un grande uso dell'astrazione; nessun progetto con milioni di righe di codice potrebbe fare altrimenti e sopravvivere. Ma l'esperienza ha dimostrato che un'eccessiva o prematura astrazione può rivelarsi dannosa al pari di una prematura ottimizzazione. L'astrazione dovrebbe essere usata fino al livello necessario e non oltre.

Ad un livello base, considerate una funzione che ha un argomento che viene sempre impostato a zero da tutti i chiamanti. Uno potrebbe mantenere quell'argomento nell'eventualità qualcuno volesse sfruttare la flessibilità offerta. In ogni caso, tuttavia, ci sono buone possibilità che il

codice che va ad implementare questo argomento aggiuntivo, sia stato rotto in maniera sottile, in un modo che non è mai stato notato - perché non è mai stato usato. Oppure, quando sorge la necessità di avere più flessibilità, questo argomento non la fornisce in maniera soddisfacente. Gli sviluppatori di Kernel, sottopongono costantemente patch che vanno a rimuovere gli argomenti inutilizzate; anche se, in generale, non avrebbero dovuto essere aggiunti.

I livelli di astrazione che nascondono l'accesso all'hardware - spesso per poter usare dei driver su diversi sistemi operativi - vengono particolarmente disapprovati. Tali livelli oscurano il codice e possono peggiorare le prestazioni; essi non appartengono al kernel Linux.

D'altro canto, se vi ritrovate a dover copiare una quantità significativa di codice proveniente da un altro sottosistema del kernel, è tempo di chiedersi se, in effetti, non avrebbe più senso togliere parte di quel codice e metterlo in una libreria separata o di implementare quella funzionalità ad un livello più elevato. Non c'è utilità nel replicare lo stesso codice per tutto il kernel.

## #ifdef e l'uso del preprocessore in generale

Il preprocessore C sembra essere una fonte di attrazione per qualche programmatore C, che ci vede una via per ottenere una grande flessibilità all'interno di un file sorgente. Ma il preprocessore non è scritto in C, e un suo massiccio impiego conduce a un codice che è molto più difficile da leggere per gli altri e che rende più difficile il lavoro di verifica del compilatore. L'uso eccessivo del preprocessore è praticamente sempre il segno di un codice che necessita di un certo lavoro di pulizia.

La compilazione condizionata con #ifdef è, in effetti, un potente strumento, ed esso viene usato all'interno del kernel. Ma esiste un piccolo desiderio: quello di vedere il codice coperto solo da una leggera spolverata di blocchi #ifdef. Come regola generale, quando possibile, l'uso di #ifdef dovrebbe essere confinato nei file d'intestazione. Il codice compilato condizionatamente può essere confinato a funzioni tali che, nel caso in cui il codice non deve essere presente, diventano vuote. Il compilatore poi ottimizzerà la chiamata alla funzione vuota rimuovendola. Il risultato è un codice molto più pulito, più facile da seguire.

Le macro del preprocessore C presentano una serie di pericoli, inclusi valutazioni multiple di espressioni che hanno effetti collaterali e non garantiscono una sicurezza rispetto ai tipi. Se siete tentati dal definire una macro, considerate l'idea di creare invece una funzione inline. Il codice che ne risulterà sarà lo stesso, ma le funzioni inline sono più leggibili, non considerano i propri argomenti più volte, e permettono al compilatore di effettuare controlli sul tipo degli argomenti e del valore di ritorno.

## Funzioni inline

Comunque, anche le funzioni inline hanno i loro pericoli. I programmatore potrebbero innamorarsi dell'efficienza percepita derivata dalla rimozione di una chiamata a funzione. Queste funzioni, tuttavia, possono ridurre le prestazioni. Dato che il loro codice viene replicato ovunque vi sia una chiamata ad esse, si finisce per gonfiare le dimensioni del kernel compilato. Questi, a turno, creano pressione sulla memoria cache del processore, e questo può causare rallentamenti importanti. Le funzioni inline, di norma, dovrebbero essere piccole e usate raramente. Il costo di una chiamata a funzione, dopo tutto, non è così alto; la creazione di molte funzioni inline è il classico esempio di un'ottimizzazione prematura.

In generale, i programmatori del kernel ignorano gli effetti della cache a loro rischio e pericolo. Il classico compromesso tempo/spazio teorizzato all'inizio delle lezioni sulle strutture dati spesso non si applica all'hardware moderno. Lo spazio è tempo, in questo senso un programma più grande sarà più lento rispetto ad uno più compatto.

I compilatori più recenti hanno preso un ruolo attivo nel decidere se una data funzione deve essere resa inline oppure no. Quindi l'uso indiscriminato della parola chiave "inline" potrebbe non essere non solo eccessivo, ma anche irrilevante.

## Sincronizzazione

Nel maggio 2006, il sistema di rete "Deviscape" fu rilasciato in pompa magna sotto la licenza GPL e reso disponibile per la sua inclusione nella ramo principale del kernel. Questa donazione fu una notizia bene accolta; il supporto per le reti senza fili era considerata, nel migliore dei casi, al di sotto degli standard; il sistema Deviscape offrì la promessa di una risoluzione a tale situazione. Tuttavia, questo codice non fu inserito nel ramo principale fino al giugno del 2007 (2.6.22). Cosa accadde?

Quel codice mostrava numerosi segnali di uno sviluppo in azienda avvenuto a porte chiuse. Ma in particolare, un grosso problema fu che non fu progettato per girare in un sistema multiprocessore. Prima che questo sistema di rete (ora chiamato mac80211) potesse essere inserito, fu necessario un lavoro sugli schemi di sincronizzazione.

Una volta, il codice del kernel Linux poteva essere sviluppato senza pensare ai problemi di concorrenza presenti nei sistemi multiprocessore. Ora, comunque, questo documento è stato scritto su di un portatile dual-core. Persino su sistemi a singolo processore, il lavoro svolto per incrementare la capacità di risposta aumenterà il livello di concorrenza interno al kernel. I giorni nei quali il codice poteva essere scritto senza pensare alla sincronizzazione sono da passati tempo.

Ogni risorsa (strutture dati, registri hardware, etc.) ai quali si potrebbe avere accesso simultaneo da più di un thread deve essere sincronizzato. Il nuovo codice dovrebbe essere scritto avendo tale accortezza in testa; riadattare la sincronizzazione a posteriori è un compito molto più difficile. Gli sviluppatori del kernel dovrebbero prendersi il tempo di comprendere bene le primitive di sincronizzazione, in modo da sceglier lo strumento corretto per eseguire un compito. Il codice che presenta una mancanza di attenzione alla concorrenza avrà un percorso difficile all'interno del ramo principale.

## Regressioni

Vale la pena menzionare un ultimo pericolo: potrebbe rivelarsi accattivante l'idea di eseguire un cambiamento (che potrebbe portare a grandi miglioramenti) che porterà ad alcune rotture per gli utenti esistenti. Questa tipologia di cambiamento è chiamata "regressione", e le regressioni sono diventate mal viste nel ramo principale del kernel. Con alcune eccezioni, i cambiamenti che causano regressioni saranno fermati se quest'ultime non potranno essere corrette in tempo utile. È molto meglio quindi evitare la regressione fin dall'inizio.

Spesso si è argomentato che una regressione può essere giustificata se essa porta risolve più problemi di quanti non ne crei. Perché, dunque, non fare un cambiamento se questo porta a nuove funzionalità a dieci sistemi per ognuno dei quali esso determina una rottura? La migliore risposta a questa domanda ci è stata fornita da Linus nel luglio 2007:

:: Dunque, noi non sistemiamo bachi introducendo nuovi problemi. Quella via nasconde insidie, e nessuno può sapere del tutto se state facendo dei progressi reali. Sono due passi avanti e uno indietro, oppure un passo avanti e due indietro?

(<http://lwn.net/Articles/243460/>).

Una particolare tipologia di regressione mal vista consiste in una qualsiasi sorta di modifica all'ABI dello spazio utente. Una volta che un'interfaccia viene esportata verso lo spazio utente, dev'essere supportata all'infinito. Questo fatto rende la creazione di interfacce per lo spazio utente particolarmente complicato: dato che non possono venir cambiate introducendo incompatibilità, esse devono essere fatte bene al primo colpo. Per questa ragione sono sempre richieste: ampie riflessioni, documentazione chiara e ampie revisioni dell'interfaccia verso lo spazio utente.

## Strumenti di verifica del codice

Almeno per ora la scrittura di codice priva di errori resta un ideale irraggiungibile ai più. Quello che speriamo di poter fare, tuttavia, è trovare e correggere molti di questi errori prima che il codice entri nel ramo principale del kernel. A tal scopo gli sviluppatori del kernel devono mettere insieme una schiera impressionante di strumenti che possano localizzare automaticamente un'ampia varietà di problemi. Qualsiasi problema trovato dal computer è un problema che non affliggerà l'utente in seguito, ne consegue che gli strumenti automatici dovrebbero essere impiegati ovunque possibile.

Il primo passo consiste semplicemente nel fare attenzione agli avvertimenti provenienti dal compilatore. Versioni moderne di gcc possono individuare (e segnalare) un gran numero di potenziali errori. Molto spesso, questi avvertimenti indicano problemi reali. Di regola, il codice inviato per la revisione non dovrebbe produrre nessun avvertimento da parte del compilatore. Per mettere a tacere gli avvertimenti, cercate di comprenderne le cause reali e cercate di evitare le "riparazioni" che fan sparire l'avvertimento senza però averne trovato la causa.

Tenete a mente che non tutti gli avvertimenti sono disabilitati di default. Costruite il kernel con "make KCFLAGS=-W" per ottenerli tutti.

Il kernel fornisce differenti opzioni che abilitano funzionalità di debugging; molti di queste sono trovano all'interno del sotto menu "kernel hacking". La maggior parte di queste opzioni possono essere attivate per qualsiasi kernel utilizzato per lo sviluppo o a scopo di test. In particolare dovreste attivare:

- FRAME\_WARN per ottenere degli avvertimenti su stack frame più grandi di un dato valore. Il risultato generato da questi avvertimenti può risultare verboso, ma non bisogna preoccuparsi per gli avvertimenti provenienti da altre parti del kernel.
- DEBUG\_OBJECTS aggiungerà un codice per tracciare il ciclo di vita di diversi oggetti creati dal kernel e avvisa quando qualcosa viene eseguito fuori controllo. Se state aggiungendo un sottosistema che crea (ed esporta) oggetti complessi propri, considerate l'aggiunta di un supporto al debugging dell'oggetto.
- DEBUG\_SLAB può trovare svariati errori di uso e di allocazione di memoria; esso dovrebbe esser usato dalla maggior parte dei kernel di sviluppo.
- DEBUG\_SPINLOCK, DEBUG\_ATOMIC\_SLEEP, e DEBUG\_MUTEXES troveranno un certo numero di errori comuni di sincronizzazione.

Esistono ancora delle altre opzioni di debugging, di alcune di esse discuteremo qui sotto. Alcune di esse hanno un forte impatto e non dovrebbero essere usate tutte le volte. Ma qualche volta il tempo speso nell'capire le opzioni disponibili porterà ad un risparmio di tempo nel breve termine.

Uno degli strumenti di debugging più tosti è il *locking checker*, o “lockdep”. Questo strumento tracerà qualsiasi acquisizione e rilascio di ogni *lock* (spinlock o mutex) nel sistema, l'ordine con il quale i *lock* sono acquisiti in relazione l'uno con l'altro, l'ambiente corrente di interruzione, eccetera. Inoltre esso può assicurare che i *lock* vengano acquisiti sempre nello stesso ordine, che le stesse assunzioni sulle interruzioni si applichino in tutte le occasioni, e così via. In altre parole, lockdep può scovare diversi scenari nei quali il sistema potrebbe, in rari casi, trovarsi in stallo. Questa tipologia di problema può essere grave (sia per gli sviluppatori che per gli utenti) in un sistema in uso; lockdep permette di trovare tali problemi automaticamente e in anticipo.

In qualità di programmatore kernel diligente, senza dubbio, dovete controllare il valore di ritorno di ogni operazione (come l'allocazione della memoria) poiché esso potrebbe fallire. Il nocciolo della questione è che i percorsi di gestione degli errori, con grande probabilità, non sono mai stati collaudati del tutto. Il codice collaudato tende ad essere codice bacato; potrete quindi essere più a vostro agio con il vostro codice se tutti questi percorsi fossero stati verificati un po' di volte.

Il kernel fornisce un framework per l'inserimento di fallimenti che fa esattamente al caso, specialmente dove sono coinvolte allocazioni di memoria. Con l'opzione per l'inserimento dei fallimenti abilitata, una certa percentuale di allocazione di memoria sarà destinata al fallimento; questi fallimenti possono essere ridotti ad uno specifico pezzo di codice. Procedere con l'inserimento dei fallimenti attivo permette al programmatore di verificare come il codice risponde quando le cose vanno male. Consultate: Documentation/fault-injection/fault-injection.rst per avere maggiori informazioni su come utilizzare questo strumento.

Altre tipologie di errori possono essere riscontrati con lo strumento di analisi statica “sparse”. Con Sparse, il programmatore può essere avvisato circa la confusione tra gli indirizzi dello spazio utente e dello spazio kernel, un miscuglio fra quantità big-endian e little-endian, il passaggio di un valore intero dove ci sia aspetta un gruppo di flag, e così via. Sparse deve essere installato separatamente (se il vostra distribuzione non lo prevede, potete trovarlo su [https://sparse.wiki.kernel.org/index.php/Main\\_Page](https://sparse.wiki.kernel.org/index.php/Main_Page)); può essere attivato sul codice aggiungendo “C=1” al comando make.

Lo strumento “Coccinelle” (<http://coccinelle.lip6.fr/>) è in grado di trovare una vasta varietà di potenziali problemi di codifica; e può inoltre proporre soluzioni per risolverli. Un buon numero di “patch semantiche” per il kernel sono state preparate nella cartella scripts/coccinelle; utilizzando “make coccicheck” esso percorrerà tali patch semantiche e farà rapporto su qualsiasi problema trovato. Per maggiori informazioni, consultate Documentation/dev-tools/coccinelle.rst.

Altri errori di portabilità sono meglio scovati compilando il vostro codice per altre architetture. Se non vi accade di avere un sistema S/390 o una scheda di sviluppo Blackfin sotto mano, potete comunque continuare la fase di compilazione. Un vasto numero di cross-compilatori per x86 possono essere trovati al sito:

<http://www.kernel.org/pub/tools/crosstool/>

Il tempo impiegato nell'installare e usare questi compilatori sarà d'aiuto nell'evitare situazioni imbarazzanti nel futuro.

## Documentazione

La documentazione è spesso stata più un'eccezione che una regola nello sviluppo del kernel. Nonostante questo, un'adeguata documentazione aiuterà a facilitare l'inserimento di nuovo codice nel kernel, rende la vita più facile per gli altri sviluppatori e sarà utile per i vostri utenti. In molti casi, la documentazione è divenuta sostanzialmente obbligatoria.

La prima parte di documentazione per qualsiasi patch è il suo changelog. Questi dovrebbero descrivere le problematiche risolte, la tipologia di soluzione, le persone che lavorano alla patch, ogni effetto rilevante sulle prestazioni e tutto ciò che può servire per la comprensione della patch. Assicuratevi che il changelog dica *perché*, vale la pena aggiungere la patch; un numero sorprendente di sviluppatori sbaglia nel fornire tale informazione.

Qualsiasi codice che aggiunge una nuova interfaccia in spazio utente - inclusi nuovi file in sysfs o /proc - dovrebbe includere la documentazione di tale interfaccia così da permettere agli sviluppatori dello spazio utente di sapere con cosa stanno lavorando. Consultate: Documentation/ABI/README per avere una descrizione di come questi documenti devono essere impostati e quali informazioni devono essere fornite.

Il file Documentation/translations/it\_IT/admin-guide/kernel-parameters.rst descrive tutti i parametri di avvio del kernel. Ogni patch che aggiunga nuovi parametri dovrebbe aggiungere nuove voci a questo file.

Ogni nuova configurazione deve essere accompagnata da un testo di supporto che spieghi chiaramente le opzioni e spieghi quando l'utente potrebbe volerle selezionare.

Per molti sottosistemi le informazioni sull'API interna sono documentate sotto forma di commenti formattati in maniera particolare; questi commenti possono essere estratti e formatati in differenti modi attraverso lo script "kernel-doc". Se state lavorando all'interno di un sottosistema che ha commenti kerneldoc dovreste mantenerli e aggiungerli, in maniera appropriata, per le funzioni disponibili esternamente. Anche in aree che non sono molto documentate, non c'è motivo per non aggiungere commenti kerneldoc per il futuro; infatti, questa può essere un'attività utile per sviluppatori novizi del kernel. Il formato di questi commenti, assieme alle informazioni su come creare modelli per kerneldoc, possono essere trovati in Documentation/translations/it\_IT/doc-guide/.

Chiunque legga un ammontare significativo di codice kernel noterà che, spesso, i commenti si fanno maggiormente notare per la loro assenza. Ancora una volta, le aspettative verso il nuovo codice sono più alte rispetto al passato; inserire codice privo di commenti sarà più difficile. Detto ciò, va aggiunto che non si desiderano commenti prolissi per il codice. Il codice dovrebbe essere, di per sé, leggibile, con dei commenti che spieghino gli aspetti più sottili.

Determinate cose dovrebbero essere sempre commentate. L'uso di barriere di memoria dovrebbero essere accompagnate da una riga che spieghi perché sia necessaria. Le regole di sincronizzazione per le strutture dati, generalmente, necessitano di una spiegazione da qualche parte. Le strutture dati più importanti, in generale, hanno bisogno di una documentazione onnicomprensiva. Le dipendenze che non sono ovvie tra bit separati di codice dovrebbero essere indicate. Tutto ciò che potrebbe indurre un inserviente del codice a fare una "pulizia" incorretta, ha bisogno di un commento che dica perché è stato fatto in quel modo. E così via.

### Cambiamenti interni dell'API

L'interfaccia binaria fornita dal kernel allo spazio utente non può essere rotta tranne che in circostanze eccezionali. L'interfaccia di programmazione interna al kernel, invece, è estremamente fluida e può essere modificata al bisogno. Se vi trovate a dover lavorare attorno ad un'API del kernel o semplicemente non state utilizzando una funzionalità offerta perché questa non rispecchia i vostri bisogni, allora questo potrebbe essere un segno che l'API ha bisogno di essere cambiata. In qualità di sviluppatore del kernel, hai il potere di fare questo tipo di modifica.

Ci sono ovviamente alcuni punti da cogliere. I cambiamenti API possono essere fatti, ma devono essere giustificati. Quindi ogni patch che porta ad una modifica dell'API interna dovrebbe essere accompagnata da una descrizione della modifica in sé e del perché essa è necessaria. Questo tipo di cambiamenti dovrebbero, inoltre, essere fatti in una patch separata, invece di essere sepolti all'interno di una patch più grande.

L'altro punto da cogliere consiste nel fatto che uno sviluppatore che modifica l'API deve, in generale, essere responsabile della correzione di tutto il codice del kernel che viene rotto per via della sua modifica. Per una funzione ampiamente usata, questo compito può condurre letteralmente a centinaia o migliaia di modifiche, molte delle quali sono in conflitto con il lavoro svolto da altri sviluppatori. Non c'è bisogno di dire che questo può essere un lavoro molto grosso, quindi è meglio essere sicuri che la motivazione sia ben solida. Notate che lo strumento Coccinelle può fornire un aiuto con modifiche estese dell'API.

Quando viene fatta una modifica API incompatibile, una persona dovrebbe, quando possibile, assicurarsi che quel codice non aggiornato sia trovato dal compilatore. Questo vi aiuterà ad essere sicuri d'avere trovato, tutti gli usi di quell'interfaccia. Inoltre questo avviserà gli sviluppatori di codice fuori dal kernel che c'è un cambiamento per il quale è necessario del lavoro. Il supporto al codice fuori dal kernel non è qualcosa di cui gli sviluppatori del kernel devono preoccuparsi, ma non dobbiamo nemmeno rendere più difficile del necessario la vita agli sviluppatori di questo codice.

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/5.Posting.rst

**Translator** Federico Vaga <[federico.vaga@vaga.pv.it](mailto:federico.vaga@vaga.pv.it)>

### Pubblicare modifiche

Prima o poi arriva il momento in cui il vostro lavoro è pronto per essere presentato alla comunità per una revisione ed eventualmente per la sua inclusione nel ramo principale del kernel. Com'era prevedibile, la comunità di sviluppo del kernel ha elaborato un insieme di convenzioni e di procedure per la pubblicazione delle patch; seguirle renderà la vita più facile a tutti quanti. Questo documento cercherà di coprire questi argomenti con un ragionevole livello di dettaglio; più informazioni possono essere trovare nella cartella 'Documentation', nei file [translations/it\\_IT/process/submitting-patches.rst](#), [translations/it\\_IT/process/submitting-drivers.rst](#), e [translations/it\\_IT/process/submit-checklist.rst](#).

## Quando pubblicarle

C'è sempre una certa resistenza nel pubblicare patch finché non sono veramente "pronte". Per semplici patch questo non è un problema. Ma quando il lavoro è di una certa complessità, c'è molto da guadagnare dai riscontri che la comunità può darvi prima che completiate il lavoro. Dovreste considerare l'idea di pubblicare un lavoro incompleto, o anche preparare un ramo git disponibile agli sviluppatori interessati, cosicché possano stare al passo col vostro lavoro in qualunque momento.

Quando pubblicate del codice che non è considerato pronto per l'inclusione, è bene che lo dicate al momento della pubblicazione. Inoltre, aggiungete informazioni sulle cose ancora da sviluppare e sui problemi conosciuti. Poche persone guarderanno delle patch che si sa essere fatte a metà, ma quelli che lo faranno penseranno di potervi aiutare a condurre il vostro sviluppo nella giusta direzione.

## Prima di creare patch

Ci sono un certo numero di cose che dovreste fare prima di considerare l'invio delle patch alla comunità di sviluppo. Queste cose includono:

- Verificare il codice fino al massimo che vi è consentito. Usate gli strumenti di debug del kernel, assicuratevi che il kernel compili con tutte le più ragionevoli combinazioni d'opzioni, usate cross-compilatori per compilare il codice per differenti architetture, eccetera.
- Assicuratevi che il vostro codice sia conforme alla linee guida del kernel sullo stile del codice.
- La vostra patch ha delle conseguenze in termini di prestazioni? Se è così, dovreste eseguire dei *benchmark* che mostrino il loro impatto (anche positivo); un riassunto dei risultati dovrebbe essere incluso nella patch.
- Siate certi d'avere i diritti per pubblicare il codice. Se questo lavoro è stato fatto per un datore di lavoro, egli avrà dei diritti su questo lavoro e dovrà quindi essere d'accordo alla sua pubblicazione con una licenza GPL

Come regola generale, pensarci un po' di più prima di inviare il codice ripaga quasi sempre lo sforzo.

## Preparazione di una patch

La preparazione delle patch per la pubblicazione può richiedere una quantità di lavoro significativa, ma, ripetiamolo ancora, generalmente sconsigliamo di risparmiare tempo in questa fase, anche sul breve periodo.

Le patch devono essere preparate per una specifica versione del kernel. Come regola generale, una patch dovrebbe basarsi sul ramo principale attuale così come lo si trova nei sorgenti git di Linus. Quando vi basate sul ramo principale, cominciate da un punto di rilascio ben noto - uno stabile o un -rc - piuttosto che creare il vostro ramo da quello principale in un punto a caso.

Per facilitare una revisione e una verifica più estesa, potrebbe diventare necessaria la produzione di versioni per -mm, linux-next o i sorgenti di un sottosistema. Basare questa patch sui suddetti sorgenti potrebbe richiedere un lavoro significativo nella risoluzione dei conflitti e nella

correzione dei cambiamenti di API; questo potrebbe variare a seconda dell'area d'interesse della vostra patch e da quello che succede altrove nel kernel.

Solo le modifiche più semplici dovrebbero essere preparate come una singola patch; tutto il resto dovrebbe essere preparato come una serie logica di modifiche. Spezzettare le patch è un po' un'arte; alcuni sviluppatori passano molto tempo nel capire come farlo in modo che piaccia alla comunità. Ci sono alcune regole spannometriche, che comunque possono aiutare considerevolmente:

- La serie di patch che pubblicherete, quasi sicuramente, non sarà come quella che trovate nel vostro sistema di controllo di versione. Invece, le vostre modifiche dovranno essere considerate nella loro forma finale, e quindi separate in parti che abbiano un senso. Gli sviluppatori sono interessati in modifiche che siano discrete e indipendenti, non alla strada che avete percorso per ottenerle.
- Ogni modifica logicamente indipendente dovrebbe essere preparata come una patch separata. Queste modifiche possono essere piccole ("aggiunto un campo in questa struttura") o grandi (l'aggiunta di un driver nuovo, per esempio), ma dovrebbero essere concettualmente piccole da permettere una descrizione in una sola riga. Ogni patch dovrebbe fare modifiche specifiche che si possano revisionare indipendentemente e di cui si possa verificare la veridicità.
- Giusto per riaffermare quanto detto sopra: non mischiate diversi tipi di modifiche nella stessa patch. Se una modifica corregge un baco critico per la sicurezza, riorganizza alcune strutture, e riformatta il codice, ci sono buone probabilità che venga ignorata e che la correzione importante venga persa.
- Ogni modifica dovrebbe portare ad un kernel che compila e funziona correttamente; se la vostra serie di patch si interrompe a metà il risultato dovrebbe essere comunque un kernel funzionante. L'applicazione parziale di una serie di patch è uno scenario comune nel quale il comando "git bisect" viene usato per trovare delle regressioni; se il risultato è un kernel guasto, renderete la vita degli sviluppatori più difficile così come quella di chi s'impegna nel nobile lavoro di scovare i problemi.
- Però, non strafate. Una volta uno sviluppatore pubblicò una serie di 500 patch che modificavano un unico file - un atto che non lo rese la persona più popolare sulla lista di discussione del kernel. Una singola patch può essere ragionevolmente grande fintanto che contenga un singolo cambiamento *logico*.
- Potrebbe essere allettante l'idea di aggiungere una nuova infrastruttura come una serie di patch, ma di lasciare questa infrastruttura inutilizzata finché l'ultima patch della serie non abilita tutto quanto. Quando è possibile, questo dovrebbe essere evitato; se questa serie aggiunge delle regressioni, "bisect" indicherà quest'ultima patch come causa del problema anche se il baco si trova altrove. Possibilmente, quando una patch aggiunge del nuovo codice dovrebbe renderlo attivo immediatamente.

Lavorare per creare la serie di patch perfetta potrebbe essere frustrante perché richiede un certo tempo e soprattutto dopo che il "vero lavoro" è già stato fatto. Quando ben fatto, comunque, è tempo ben speso.

## Formattazione delle patch e i changelog

Quindi adesso avete una serie perfetta di patch pronte per la pubblicazione, ma il lavoro non è davvero finito. Ogni patch deve essere preparata con un messaggio che spieghi al resto del mondo, in modo chiaro e veloce, il suo scopo. Per ottenerlo, ogni patch sarà composta dai seguenti elementi:

- Un campo opzionale “From” col nome dell’autore della patch. Questa riga è necessaria solo se state passando la patch di qualcun altro via email, ma nel dubbio non fa di certo male aggiungerlo.
- Una descrizione di una riga che spieghi cosa fa la patch. Questo messaggio dovrebbe essere sufficiente per far comprendere al lettore lo scopo della patch senza altre informazioni. Questo messaggio, solitamente, presenta in testa il nome del sottosistema a cui si riferisce, seguito dallo scopo della patch. Per esempio:

```
gpio: fix build on CONFIG_GPIO_SYSFS=n
```

- Una riga bianca seguita da una descrizione dettagliata della patch. Questa descrizione può essere lunga tanto quanto serve; dovrebbe spiegare cosa fa e perché dovrebbe essere aggiunta al kernel.
- Una o più righe etichette, con, minimo, una riga *Signed-off-by*: col nome dall’autore della patch. Queste etichette verranno descritte meglio più avanti.

Gli elementi qui sopra, assieme, formano il changelog di una patch. Scrivere un buon changelog è cruciale ma è spesso un’arte trascurata; vale la pena spendere qualche parola in più al riguardo. Quando scrivete un changelog dovrete tenere ben presente che molte persone leggeranno le vostre parole. Queste includono i manutentori di un sotto-sistema, e i revisori che devono decidere se la patch debba essere inclusa o no, le distribuzioni e altri manutentori che cercano di valutare se la patch debba essere applicata su kernel più vecchi, i cacciatori di bachi che si chiederanno se la patch è la causa di un problema che stanno cercando, gli utenti che vogliono sapere com’è cambiato il kernel, e molti altri. Un buon changelog fornisce le informazioni necessarie a tutte queste persone nel modo più diretto e conciso possibile.

A questo scopo, la riga riassuntiva dovrebbe descrivere gli effetti della modifica e la motivazione della patch nel modo migliore possibile nonostante il limite di una sola riga. La descrizione dettagliata può spiegare meglio i temi e fornire maggiori informazioni. Se una patch corregge un baco, citate, se possibile, il commit che lo introduce (e per favore, quando citate un commit aggiungete sia il suo identificativo che il titolo). Se il problema è associabile ad un file di log o all’output del compilatore, includeteli al fine d’aiutare gli altri a trovare soluzioni per lo stesso problema. Se la modifica ha lo scopo di essere di supporto a sviluppi successivi, ditelo. Se le API interne vengono cambiate, dettagliate queste modifiche e come gli altri dovrebbero agire per applicarle. In generale, più riuscirete ad entrare nei panni di tutti quelli che leggeranno il vostro changelog, meglio sarà il changelog (e il kernel nel suo insieme).

Non serve dirlo, un changelog dovrebbe essere il testo usato nel messaggio di commit in un sistema di controllo di versione. Sarà seguito da:

- La patch stessa, nel formato unificato per patch (“-u”). Usare l’opzione “-p” assocerà alla modifica il nome della funzione alla quale si riferisce, rendendo il risultato più facile da leggere per gli altri.

Dovreste evitare di includere nelle patch delle modifiche per file irrilevanti (quelli generati dal processo di generazione, per esempio, o i file di backup del vostro editor). Il file “dontdiff” nella

cartella Documentation potrà esservi d'aiuto su questo punto; passatelo a diff con l'opzione “-X”.

Le etichette sopra menzionante sono usate per descrivere come i vari sviluppatori sono stati associati allo sviluppo di una patch. Sono descritte in dettaglio nel documento [translations/it\\_IT/process/submitting-patches.rst](#); quello che segue è un breve riassunto. Ognuna di queste righe ha il seguente formato:

```
tag: Full Name <email address> optional-other-stuff
```

Le etichette in uso più comuni sono:

- Signed-off-by: questa è la certificazione che lo sviluppatore ha il diritto di sottomettere la patch per l'integrazione nel kernel. Questo rappresenta il consenso verso il certificato d'origine degli sviluppatori, il testo completo potrà essere trovato in [Documentation/translations/it\\_IT/process/submitting-patches.rst](#). Codice che non presenta una firma appropriata non potrà essere integrato.
- Co-developed-by: indica che la patch è stata cosviluppata da diversi sviluppatori; viene usato per assegnare più autori (in aggiunta a quello associato all'etichetta From:) quando più persone lavorano ad una patch. Ogni Co-developed-by: dev'essere seguito immediatamente da un Signed-off-by: del corrispondente coautore. Maggiori dettagli ed esempi sono disponibili in [Documentation/translations/it\\_IT/process/submitting-patches.rst](#).
- Acked-by: indica il consenso di un altro sviluppatore (spesso il manutentore del codice in oggetto) all'integrazione della patch nel kernel.
- Tested-by: menziona la persona che ha verificato la patch e l'ha trovata funzionante.
- Reviewed-by: menziona lo sviluppatore che ha revisionato la patch; per maggiori dettagli leggete la dichiarazione dei revisori in [Documentation/translations/it\\_IT/process/submitting-patches.rst](#)
- Reported-by: menziona l'utente che ha riportato il problema corretto da questa patch; quest'etichetta viene usata per dare credito alle persone che hanno verificato il codice e ci hanno fatto sapere quando le cose non funzionavano correttamente.
- Cc: la persona menzionata ha ricevuto una copia della patch ed ha avuto l'opportunità di commentarla.

State attenti ad aggiungere queste etichette alla vostra patch: solo “Cc:” può essere aggiunta senza il permesso esplicito della persona menzionata.

### Inviare la modifica

Prima di inviare la vostra patch, ci sarebbero ancora un paio di cose di cui dovreste aver cura:

- Siete sicuri che il vostro programma di posta non corromperà le patch? Le patch che hanno spazi bianchi in libertà o andate a capo aggiunti dai programmi di posta non funzioneranno per chi le riceve, e spesso non verranno nemmeno esaminate in dettaglio. Se avete un qualsiasi dubbio, inviate la patch a voi stessi e verificate che sia integra.

[Documentation/translations/it\\_IT/process/email-clients.rst](#) contiene alcuni suggerimenti utili sulla configurazione dei programmi di posta al fine di inviare patch.

- Siete sicuri che la vostra patch non contenga sciocchi errori? Dovreste sempre processare le patch con scripts/checkpatch.pl e correggere eventuali problemi riportati. Per favore

tenete ben presente che checkpatch.pl non è più intelligente di voi, nonostante sia il risultato di un certa quantità di ragionamenti su come debba essere una patch per il kernel. Se seguire i suggerimenti di checkpatch.pl rende il codice peggiore, allora non fatelo.

Le patch dovrebbero essere sempre inviate come testo puro. Per favore non inviatele come allegati; questo rende molto più difficile, per i revisori, citare parti della patch che si vogliono commentare. Invece, mettete la vostra patch direttamente nel messaggio.

Quando inviate le patch, è importante inviarne una copia a tutte le persone che potrebbero esserne interessate. Al contrario di altri progetti, il kernel incoraggia le persone a peccare nell'invio di tante copie; non presumente che le persone interessate vedano i vostri messaggi sulla lista di discussione. In particolare le copie dovrebbero essere inviate a:

- I manutentori dei sottosistemi affetti della modifica. Come descritto in precedenza, il file `MAINTAINERS` è il primo luogo dove cercare i nomi di queste persone.
- Altri sviluppatori che hanno lavorato nello stesso ambiente - specialmente quelli che potrebbero lavorarci proprio ora. Usate git potrebbe essere utile per vedere chi altri ha modificato i file su cui state lavorando.
- Se state rispondendo a un rapporto su un baco, o a una richiesta di funzionalità, includete anche gli autori di quei rapporti/richieste.
- Inviate una copia alle liste di discussione interessate, o, se nient'altro è adatto, alla lista `linux-kernel`
- Se state correggendo un baco, pensate se la patch dovrebbe essere inclusa nel prossimo rilascio stabile. Se è così, la lista di discussione `stable@vger.kernel.org` dovrebbe riceverne una copia. Aggiungete anche l'etichetta "Cc: `stable@vger.kernel.org`" nella patch stessa; questo permetterà alla squadra `stable` di ricevere una notifica quando questa correzione viene integrata nel ramo principale.

Quando scegliete i destinatari della patch, è bene avere un'idea di chi pensiate che sia colui che, eventualmente, accetterà la vostra patch e la integrerà. Nonostante sia possibile inviare patch direttamente a Linus Torvalds, e lasciare che sia lui ad integrarle, solitamente non è la strada migliore da seguire. Linus è occupato, e ci sono dei manutentori di sotto-sistema che controllano una parte specifica del kernel. Solitamente, vorreste che siano questi manutentori ad integrare le vostre patch. Se non c'è un chiaro manutentore, l'ultima spiaggia è spesso Andrew Morton.

Le patch devono avere anche un buon oggetto. Il tipico formato per l'oggetto di una patch assomiglia a questo:

[PATCH nn/mm] subsys: one-line description of the patch

dove "nn" è il numero ordinale della patch, "mm" è il numero totale delle patch nella serie, e "subsys" è il nome del sottosistema interessato. Chiaramente, nn/mm può essere omesso per una serie composta da una singola patch.

Se avete una significativa serie di patch, è prassi inviare una descrizione introduttiva come parte zero. Tuttavia questa convenzione non è universalmente seguita; se la usate, ricordate che le informazioni nell'introduzione non faranno parte del changelog del kernel. Quindi per favore, assicuratevi che ogni patch abbia un changelog completo.

In generale, la seconda parte e quelle successive di una patch "composta" dovrebbero essere inviate come risposta alla prima, cosicché vengano viste come un unico *thread*. Strumenti come git e quilt hanno comandi per inviare gruppi di patch con la struttura appropriata. Se avete una

serie lunga e state usando git, per favore state alla larga dall'opzione `-chain-reply-to` per evitare di creare un annidamento eccessivo.

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/6.Followthrough.rst

**Translator** Alessia Mantegazza <[amantegazza@vaga.pv.it](mailto:amantegazza@vaga.pv.it)>

### Completamento

A questo punto, avete seguito le linee guida fino a questo punto e, con l'aggiunta delle vostre capacità ingegneristiche, avete pubblicato una serie perfetta di patch. Uno dei più grandi errori che possono essere commessi persino da sviluppatori kernel esperti è quello di concludere che il lavoro sia ormai finito. In verità, la pubblicazione delle patch simboleggia una transizione alla fase successiva del processo, con, probabilmente, ancora un po' di lavoro da fare.

È raro che una modifica sia così bella alla sua prima pubblicazione che non ci sia alcuno spazio di miglioramento. Il programma di sviluppo del kernel riconosce questo fatto e quindi, è fortemente orientato al miglioramento del codice pubblicato. Voi, in qualità di autori del codice, dovete lavorare con la comunità del kernel per assicurare che il vostro codice mantenga gli standard qualitativi richiesti. Un fallimento in questo processo è quasi come impedire l'inclusione delle vostre patch nel ramo principale.

### Lavorare con i revisori

Una patch che abbia una certa rilevanza avrà ricevuto numerosi commenti da parte di altri sviluppatori dato che avranno revisionato il codice. Lavorare con i revisori può rivelarsi, per molti sviluppatori, la parte più intimidatoria del processo di sviluppo del kernel. La vita può esservi resa molto più facile se tenete presente alcuni dettagli:

- Se avete descritto la vostra modifica correttamente, i revisori ne comprenderanno il valore e il perché vi siete presi il disturbo di scriverla. Ma tale valore non li tratterà dal porvi una domanda fondamentale: come verrà mantenuto questo codice nel kernel nei prossimi cinque o dieci anni? Molti dei cambiamenti che potrebbero esservi richiesti - da piccoli problemi di stile a sostanziali ristesure - vengono dalla consapevolezza che Linux resterà in circolazione e in continuo sviluppo ancora per diverse decadi.
- La revisione del codice è un duro lavoro, ed è un mestiere poco riconosciuto; le persone ricordano chi ha scritto il codice, ma meno fama è attribuita a chi lo ha revisionato. Quindi i revisori potrebbero divenire burberi, specialmente quando vendono i medesimi errori venire fatti ancora e ancora. Se ricevete una revisione che vi sembra abbia un tono arrabbiato, insultante o addirittura offensivo, resistente alla tentazione di rispondere a tono. La revisione riguarda il codice e non la persona, e i revisori non vi stanno attaccando personalmente.
- Similmente, i revisori del codice non stanno cercando di promuovere i loro interessi a vostre spese. Gli sviluppatori del kernel spesso si aspettano di lavorare sul kernel per anni, ma sanno che il loro datore di lavoro può cambiare. Davvero, senza praticamente

eccezioni, loro stanno lavorando per la creazione del miglior kernel possibile; non stanno cercando di creare un disagio ad aziende concorrenti.

Quello che si sta cercando di dire è che, quando i revisori vi inviano degli appunti dovete fare attenzione alle osservazioni tecniche che vi stanno facendo. Non lasciate che il loro modo di esprimersi o il vostro orgoglio impediscano che ciò accada. Quando avete dei suggerimenti sulla revisione, prendetevi il tempo per comprendere cosa il revisore stia cercando di comunicarvi. Se possibile, sistematate le cose che il revisore vi chiede di modificare. E rispondete al revisore ringraziandolo e spiegando come intendete fare.

Notate che non dovete per forza essere d'accordo con ogni singola modifica suggerita dai revisori. Se credete che il revisore non abbia compreso il vostro codice, spiegateglielo. Se avete un'obiezione tecnica da fargli su di una modifica suggerita, spiegateela inserendo anche la vostra soluzione al problema. Se la vostra spiegazione ha senso, il revisore la accetterà. Tuttavia, la vostra motivazione potrebbe non essere del tutto persuasiva, specialmente se altri iniziano ad essere d'accordo con il revisore. Prendetevi quindi un po' di tempo per pensare ancora alla cosa. Può risultare facile essere accecati dalla propria soluzione al punto che non realizzate che c'è qualcosa di fondamentalmente sbagliato o, magari, non state nemmeno risolvendo il problema giusto.

Andrew Morton suggerisce che ogni suggerimento di revisione che non è presente nella modifica del codice dovrebbe essere inserito in un commento aggiuntivo; ciò può essere d'aiuto ai futuri revisori nell'evitare domande che sorgono al primo sguardo.

Un errore fatale è quello di ignorare i commenti di revisione nella speranza che se ne andranno. Non andranno via. Se pubblicherete nuovamente il codice senza aver risposto ai commenti ricevuti, probabilmente le vostre modifiche non andranno da nessuna parte.

Parlando di ripubblicazione del codice: per favore tenete a mente che i revisori non ricorderanno tutti i dettagli del codice che avete pubblicato l'ultima volta. Quindi è sempre una buona idea quella di ricordare ai revisori le questioni sollevate precedentemente e come le avete risolte. I revisori non dovrebbero star lì a cercare all'interno degli archivi per familiarizzare con ciò che è stato detto l'ultima volta; se li aiutate in questo senso, saranno di umore migliore quando riguarderanno il vostro codice.

Se invece avete cercato di far tutto correttamente ma le cose continuano a non andar bene? Molti disaccordi di natura tecnica possono essere risolti attraverso la discussione, ma ci sono volte dove qualcuno deve prendere una decisione. Se credete veramente che tale decisione andrà contro di voi ingiustamente, potete sempre tentare di rivolgervi a qualcuno più in alto di voi. Per cose di questo genere la persona con più potere è Andrew Morton. Andrew è una figura molto rispettata all'interno della comunità di sviluppo del kernel; lui può spesso sbrogliare situazioni che sembrano irrimediabilmente bloccate. Rivolgersi ad Andrew non deve essere fatto alla leggera, e non deve essere fatto prima di aver esplorato tutte le altre alternative. E tenete a mente, ovviamente, che nemmeno lui potrebbe non essere d'accordo con voi.

### Cosa accade poi

Se la modifica è ritenuta un elemento valido da essere aggiunta al kernel, e una volta che la maggior parte degli appunti dei revisori sono stati sistemati, il passo successivo solitamente è quello di entrare in un sottosistema gestito da un manutentore. Come ciò avviene dipende dal sottosistema medesimo; ogni manutentore ha il proprio modo di fare le cose. In particolare, ci potrebbero essere diversi sorgenti - uno, magari, dedicato alle modifiche pianificate per la finestra di fusione successiva, e un altro per il lavoro di lungo periodo.

Per le modifiche proposte in aree per le quali non esiste un sottosistema preciso (modifiche di gestione della memoria, per esempio), i sorgenti di ripiego finiscono per essere -mm. Ed anche le modifiche che riguardano più sottosistemi possono finire in quest'ultimo.

L'inclusione nei sorgenti di un sottosistema può comportare per una patch, un alto livello di visibilità. Ora altri sviluppatori che stanno lavorando in quei medesimi sorgenti avranno le vostre modifiche. I sottosistemi solitamente riforniscono anche Linux-next, rendendo i propri contenuti visibili all'intera comunità di sviluppo. A questo punto, ci sono buone possibilità per voi di ricevere ulteriori commenti da un nuovo gruppo di revisori; anche a questi commenti dovete rispondere come avete già fatto per gli altri.

Ciò che potrebbe accadere a questo punto, in base alla natura della vostra modifica, riguarda eventuali conflitti con il lavoro svolto da altri. Nella peggiore delle situazioni, i conflitti più pesanti tra modifiche possono concludersi con la messa a lato di alcuni dei lavori svolti cosicché le modifiche restanti possano funzionare ed essere integrate. Altre volte, la risoluzione dei conflitti richiederà del lavoro con altri sviluppatori e, possibilmente, lo spostamento di alcune patch da dei sorgenti a degli altri in modo da assicurare che tutto sia applicato in modo pulito. Questo lavoro può rivelarsi una spina nel fianco, ma consideratevi fortunati: prima dell'avvento dei sorgenti linux-next, questi conflitti spesso emergevano solo durante l'apertura della finestra di integrazione e dovevano essere smaltiti in fretta. Ora essi possono essere risolti comodamente, prima dell'apertura della finestra.

Un giorno, se tutto va bene, vi collegherete e vedrete che la vostra patch è stata inserita nel ramo principale de kernel. Congratulazioni! Terminati i festeggiamenti (nel frattempo avrete inserito il vostro nome nel file MAINTAINERS) vale la pena ricordare una piccola cosa, ma importante: il lavoro non è ancora finito. L'inserimento nel ramo principale porta con se nuove sfide.

Cominciamo con il dire che ora la visibilità della vostra modifica è ulteriormente cresciuta. Ci potrebbe portare ad una nuova fase di commenti dagli sviluppatori che non erano ancora a conoscenza della vostra patch. Ignorarli potrebbe essere allettante dato che non ci sono più dubbi sull'integrazione della modifica. Resistete a tale tentazione, dovete mantenervi disponibili agli sviluppatori che hanno domande o suggerimenti per voi.

Ancora più importante: l'inclusione nel ramo principale mette il vostro codice nelle mani di un gruppo di *tester* molto più esteso. Anche se avete contribuito ad un driver per un hardware che non è ancora disponibile, sarete sorpresi da quante persone inseriranno il vostro codice nei loro kernel. E, ovviamente, dove ci sono *tester*, ci saranno anche dei rapporti su eventuali bachi.

La peggior specie di rapporti sono quelli che indicano delle regressioni. Se la vostra modifica causa una regressione, avrete un gran numero di occhi puntati su di voi; la regressione deve essere sistemata il prima possibile. Se non vorrete o non sarete capaci di sistemerla (e nessuno lo farà per voi), la vostra modifica sarà quasi certamente rimossa durante la fase di stabilizzazione. Oltre alla perdita di tutto il lavoro svolto per far sì che la vostra modifica fosse

inserita nel ramo principale, l'avere una modifica rimossa a causa del fallimento nel sistemare una regressione, potrebbe rendere più difficile per voi far accettare il vostro lavoro in futuro.

Dopo che ogni regressione è stata affrontata, ci potrebbero essere altri bachi ordinari da “sconfiggere”. Il periodo di stabilizzazione è la vostra migliore opportunità per sistemare questi bachi e assicurarvi che il debutto del vostro codice nel ramo principale del kernel sia il più solido possibile. Quindi, per favore, rispondete ai rapporti sui bachi e ponete rimedio, se possibile, a tutti i problemi. È a questo che serve il periodo di stabilizzazione; potete iniziare creando nuove fantastiche modifiche una volta che ogni problema con le vecchie sia stato risolto.

Non dimenticate che esistono altre pietre miliari che possono generare rapporti sui bachi: il successivo rilascio stabile, quando una distribuzione importante usa una versione del kernel nel quale è presente la vostra modifica, eccetera. Il continuare a rispondere a questi rapporti è fonte di orgoglio per il vostro lavoro. Se questa non è una sufficiente motivazione, allora, è anche consigliabile considerare che la comunità di sviluppo ricorda gli sviluppatori che hanno perso interesse per il loro codice una volta integrato. La prossima volta che pubblicherete una patch, la comunità la valuterà anche sulla base del fatto che non sarete disponibili a prendervene cura anche nel futuro.

## Altre cose che posso accadere

Un giorno, potrete aprire la vostra email e vedere che qualcuno vi ha inviato una patch per il vostro codice. Questo, dopo tutto, è uno dei vantaggi di avere il vostro codice “là fuori”. Se siete d'accordo con la modifica, potrete anche inoltrarla ad un manutentore di sottosistema (assicuratevi di includere la riga “From:” cosicché l'attribuzione sia corretta, e aggiungete una vostra firma “Signed-off-by”), oppure inviate un “Acked-by:” e lasciate che l'autore originale la invii.

Se non siete d'accordo con la patch, inviate una risposta educata spiegando il perché. Se possibile, dite all'autore quali cambiamenti servirebbero per rendere la patch accettabile da voi. C'è una certa riluttanza nell'inserire modifiche con un conflitto fra autore e manutentore del codice, ma solo fino ad un certo punto. Se siete visti come qualcuno che blocca un buon lavoro senza motivo, quelle patch vi passeranno oltre e andranno nel ramo principale in ogni caso. Nel kernel Linux, nessuno ha potere di voto assoluto su alcun codice. Eccezione fatta per Linus, forse.

In rarissime occasioni, potrete vedere qualcosa di completamente diverso: un altro sviluppatore che pubblica una soluzione differente al vostro problema. A questo punto, c'è una buona probabilità che una delle due modifiche non verrà integrata, e il “c'ero prima io” non è considerato un argomento tecnico rilevante. Se la modifica di qualcun'altro rimpiazza la vostra ed entra nel ramo principale, esiste un unico modo di reagire: siate contenti che il vostro problema sia stato risolto e andate avanti con il vostro lavoro. L'avere un vostro lavoro spintonato da parte in questo modo può essere avvilente e scoraggiante, ma la comunità ricorderà come avrete reagito anche dopo che avrà dimenticato quale fu la modifica accettata.

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/7.AdvancedTopics.rst

**Translator** Federico Vaga <[federico.vaga@vaga.pv.it](mailto:federico.vaga@vaga.pv.it)>

### Argomenti avanzati

A questo punto, si spera, dovreste avere un'idea su come funziona il processo di sviluppo. Ma rimane comunque molto da imparare! Questo capitolo copre alcuni argomenti che potrebbero essere utili per gli sviluppatori che stanno per diventare parte integrante del processo di sviluppo del kernel.

### Gestire le modifiche con git

L'uso di un sistema distribuito per il controllo delle versioni del kernel ebbe inizio nel 2002 quando Linux iniziò a provare il programma proprietario BitKeeper. Nonostante l'uso di BitKeeper fosse opinabile, di certo il suo approccio alla gestione dei sorgenti non lo era. Un sistema distribuito per il controllo delle versioni accelerò immediatamente lo sviluppo del kernel. Oggi, ci sono diverse alternative libere a BitKeeper. Per il meglio o il peggio, il progetto del kernel ha deciso di usare git per gestire i sorgenti.

Gestire le modifiche con git può rendere la vita dello sviluppatore molto più facile, specialmente quando il volume delle modifiche cresce. Git ha anche i suoi lati taglienti che possono essere pericolosi; è uno strumento giovane e potente che è ancora in fase di civilizzazione da parte dei suoi sviluppatori. Questo documento non ha lo scopo di insegnare l'uso di git ai suoi lettori; ci sarebbe materiale a sufficienza per un lungo documento al riguardo. Invece, qui ci concentriamo in particolare su come git è parte del processo di sviluppo del kernel. Gli sviluppatori che desiderassero diventare agili con git troveranno più informazioni ai seguenti indirizzi:

<http://git-scm.com/>

<http://www.kernel.org/pub/software/scm/git/docs/user-manual.html>

e su varie guide che potrete trovare su internet.

La prima cosa da fare prima di usarlo per produrre patch che saranno disponibili ad altri, è quella di leggere i siti qui sopra e di acquisire una base solida su come funziona git. Uno sviluppatore che sappia usare git dovrebbe essere capace di ottenere una copia del repository principale, esplorare la storia della revisione, registrare le modifiche, usare i rami, eccetera. Una certa comprensione degli strumenti git per riscrivere la storia (come rebase) è altrettanto utile. Git ha i propri concetti e la propria terminologia; un nuovo utente dovrebbe conoscere *refs*, *remote branch*, *index*, *fast-forward merge*, *push* e *pull*, *detached head*, eccetera. Il tutto potrebbe essere un po' intimidatorio visto da fuori, ma con un po' di studio i concetti non saranno così difficili da capire.

Utilizzare git per produrre patch da sottomettere via email può essere un buon esercizio da fare mentre si sta prendendo confidenza con lo strumento.

Quando sarete in grado di creare rami git che siano guardabili da altri, vi servirà, ovviamente, un server dal quale sia possibile attingere le vostre modifiche. Se avete un server accessibile da Internet, configurarlo per eseguire git-daemon è relativamente semplice. Altrimenti, iniziano a svilupparsi piattaforme che offrono spazi pubblici, e gratuiti (Github, per esempio). Gli sviluppatori permanenti possono ottenere un account su kernel.org, ma non è proprio facile da ottenere; per maggiori informazioni consultate la pagina web <http://kernel.org/faq/>.

In git è normale avere a che fare con tanti rami. Ogni linea di sviluppo può essere separata in "rami per argomenti" e gestiti indipendentemente. In git i rami sono facilissimi, per cui non c'è motivo per non usarli in libertà. In ogni caso, non dovreste sviluppare su alcun ramo dal quale altri potrebbero attingere. I rami disponibili pubblicamente dovrebbero essere creati con

attenzione; integrate patch dai rami di sviluppo solo quando sono complete e pronte ad essere consegnate - non prima.

Git offre alcuni strumenti che vi permettono di riscrivere la storia del vostro sviluppo. Una modifica errata (diciamo, una che rompe la bisezione, oppure che ha un qualche tipo di baco evidente) può essere corretta sul posto o fatta sparire completamente dalla storia. Una serie di patch può essere riscritta come se fosse stata scritta in cima al ramo principale di oggi, anche se ci avete lavorato per mesi. Le modifiche possono essere spostate in modo trasparente da un ramo ad un altro. E così via. Un uso giudizioso di git per revisionare la storia può aiutare nella creazione di una serie di patch pulite e con meno problemi.

Un uso eccessivo può portare ad altri tipi di problemi, tuttavia, oltre alla semplice ossessione per la creazione di una storia del progetto che sia perfetta. Riscrivere la storia riscriverà le patch contenute in quella storia, trasformando un kernel verificato (si spera) in uno da verificare. Ma, oltre a questo, gli sviluppatori non possono collaborare se non condividono la stessa vista sulla storia del progetto; se riscrivete la storia dalla quale altri sviluppatori hanno attinto per i loro repository, renderete la loro vita molto più difficile. Quindi tenete conto di questa semplice regola generale: la storia che avete esposto ad altri, generalmente, dovrebbe essere vista come immutabile.

Dunque, una volta che il vostro insieme di patch è stato reso disponibile pubblicamente non dovrebbe essere più sovrascritto. Git tenterà di imporre questa regola, e si rifiuterà di pubblicare nuove patch che non risultino essere dirette discendenti di quelle pubblicate in precedenza (in altre parole, patch che non condividono la stessa storia). È possibile ignorare questo controllo, e ci saranno momenti in cui sarà davvero necessario riscrivere un ramo già pubblicato. Un esempio è linux-next dove le patch vengono spostate da un ramo all'altro al fine di evitare conflitti. Ma questo tipo d'azione dovrebbe essere un'eccezione. Questo è uno dei motivi per cui lo sviluppo dovrebbe avvenire in rami privati (che possono essere sovrascritti quando lo si ritiene necessario) e reso pubblico solo quando è in uno stato avanzato.

Man mano che il ramo principale (o altri rami su cui avete basato le modifiche) avanza, diventa allettante l'idea di integrare tutte le patch per rimanere sempre aggiornati. Per un ramo privato, il *rebase* può essere un modo semplice per rimanere aggiornati, ma questa non è un'opzione nel momento in cui il vostro ramo è stato esposto al mondo intero. *Merge* occasionali possono essere considerati di buon senso, ma quando diventano troppo frequenti confondono inutilmente la storia. La tecnica suggerita in questi casi è quella di fare *merge* raramente, e più in generale solo nei momenti di rilascio (per esempio gli -rc del ramo principale). Se siete nervosi circa alcune patch in particolare, potete sempre fare dei *merge* di test in un ramo privato. In queste situazioni git "rerere" può essere utile; questo strumento si ricorda come i conflitti di *merge* furono risolti in passato cosicché non dovete fare lo stesso lavoro due volte.

Una delle lamentele più grosse e ricorrenti sull'uso di strumenti come git è il grande movimento di patch da un repository all'altro che rende facile l'integrazione nel ramo principale di modifiche mediocri, il tutto sotto il naso dei revisori. Gli sviluppatori del kernel tendono ad essere scontenti quando vedono succedere queste cose; preparare un ramo git con patch che non hanno ricevuto alcuna revisione o completamente avulse, potrebbe influire sulla vostra capacità di proporre, in futuro, l'integrazione dei vostri rami. Citando Linus

Potete inviarmi le vostre patch, ma per far sì che io integri una vostra modifica da git, devo sapere che voi sappiate cosa state facendo, e ho bisogno di fidarmi \*senza\* dover passare tutte le modifiche manualmente una per una.

(<http://lwn.net/Articles/224135/>).

Per evitare queste situazioni, assicuratevi che tutte le patch in un ramo siano strettamente correlate al tema delle modifiche; un ramo “driver fixes” non dovrebbe fare modifiche al codice principale per la gestione della memoria. E, più importante ancora, non usate un repository git per tentare di evitare il processo di revisione. Pubblicate un sommario di quello che il vostro ramo contiene sulle liste di discussione più opportune, e, quando sarà il momento, richiedete che il vostro ramo venga integrato in linux-next.

Se e quando altri inizieranno ad inviarvi patch per essere incluse nel vostro repository, non dovete dimenticare di revisionarle. Inoltre assicuratevi di mantenerne le informazioni di paternità; al riguardo git “am” fa del suo meglio, ma potreste dover aggiungere una riga “From:” alla patch nel caso in cui sia arrivata per vie traverse.

Quando richiedete l’integrazione, siate certi di fornire tutte le informazioni: dov’è il vostro repository, quale ramo integrare, e quali cambiamenti si otterranno dall’integrazione. Il comando git request-pull può essere d’aiuto; preparerà una richiesta nel modo in cui gli altri sviluppatori se l’aspettano, e verificherà che vi siate ricordati di pubblicare quelle patch su un server pubblico.

### Revisionare le patch

Alcuni lettori potrebbero avere obiezioni sulla presenza di questa sezione negli “argomenti avanzati” sulla base che anche gli sviluppatori principianti dovrebbero revisionare le patch. È certamente vero che non c’è modo migliore di imparare come programmare per il kernel che guardare il codice pubblicato dagli altri. In aggiunta, i revisori sono sempre troppo pochi; guardando il codice potete apportare un significativo contributo all’intero processo.

Revisionare il codice potrebbe risultare intimidatorio, specialmente per i nuovi arrivati che potrebbero sentirsi un po’ nervosi nel questionare il codice - in pubblico - pubblicato da sviluppatori più esperti. Perfino il codice scritto dagli sviluppatori più esperti può essere migliorato. Forse il suggerimento migliore per i revisori (tutti) è questo: formulate i commenti come domande e non come critiche. Chiedere “Come viene rilasciato il *lock* in questo percorso?” funziona sempre molto meglio che “qui la sincronizzazione è sbagliata”.

Diversi sviluppatori revisioneranno il codice con diversi punti di vista. Alcuni potrebbero concentrarsi principalmente sullo stile del codice e se alcune linee hanno degli spazio bianchi di troppo. Altri si chiederanno se accettare una modifica interamente è una cosa positiva per il kernel o no. E altri ancora si focalizzeranno sui problemi di sincronizzazione, l’uso eccessivo di *stack*, problemi di sicurezza, duplicazione del codice in altri contesti, documentazione, effetti negativi sulle prestazioni, cambi all’ABI dello spazio utente, eccetera. Qualunque tipo di revisione è ben accetta e di valore, se porta ad avere un codice migliore nel kernel.

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l’unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/8.Conclusion.rst

**Translator** Alessia Mantegazza <[amanantegazza@vaga.pv.it](mailto:amanantegazza@vaga.pv.it)>

## Per maggiori informazioni

Esistono numerose fonti di informazioni sullo sviluppo del kernel Linux e argomenti correlati. Primo tra questi sarà sempre la cartella Documentation che si trova nei sorgenti kernel.

Il file *process/howto.rst* è un punto di partenza importante; *processSubmitting-patches.rst* e *processSubmitting-drivers.rst* sono anch'essi qualcosa che tutti gli sviluppatori del kernel dovrebbero leggere. Molte API interne al kernel sono documentate utilizzando il meccanismo kerneldoc; "make htmldocs" o "make pdfdocs" possono essere usati per generare quei documenti in HTML o PDF (sebbene le versioni di TeX di alcune distribuzioni hanno dei limiti interni e fallisce nel processare appropriatamente i documenti).

Diversi siti web approfondiscono lo sviluppo del kernel ad ogni livello di dettaglio. Il vostro autore vorrebbe umilmente suggerirvi <http://lwn.net/> come fonte; usando l'indice 'kernel' su LWN troverete molti argomenti specifici sul kernel:

<http://lwn.net/Kernel/Index/>

Oltre a ciò, una risorsa valida per gli sviluppatori kernel è:

<http://kernelnewbies.org/>

E, ovviamente, una fonte da non dimenticare è <http://kernel.org/>, il luogo definitivo per le informazioni sui rilasci del kernel.

Ci sono numerosi libri sullo sviluppo del kernel:

Linux Device Drivers, 3rd Edition (Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman). In linea all'indirizzo <http://lwn.net/Kernel/LDD3/>.

Linux Kernel Development (Robert Love).

Understanding the Linux Kernel (Daniel Bovet and Marco Cesati).

Tutti questi libri soffrono di un errore comune: tendono a risultare in un certo senso obsoleti dal momento che si trovano in libreria da diverso tempo. Comunque contengono informazioni abbastanza buone.

La documentazione per git la troverete su:

<http://www.kernel.org/pub/software/scm/git/docs/>

<http://www.kernel.org/pub/software/scm/git/docs/user-manual.html>

## Conclusioni

Congratulazioni a chiunque ce l'abbia fatta a terminare questo documento di lungo-respiro. Si spera che abbia fornito un'utile comprensione d'insieme di come il kernel Linux viene sviluppato e di come potete partecipare a tale processo.

Infine, quello che conta è partecipare. Qualsiasi progetto software open-source non è altro che la somma di quello che i suoi contributori mettono al suo interno. Il kernel Linux è cresciuto velocemente e bene perché ha ricevuto il supporto di un impressionante gruppo di sviluppatori, ognuno dei quali sta lavorando per renderlo migliore. Il kernel è un esempio importante di cosa può essere fatto quando migliaia di persone lavorano insieme verso un obiettivo comune.

Il kernel può sempre beneficiare di una larga base di sviluppatori, tuttavia, c'è sempre molto lavoro da fare. Ma, cosa non meno importante, molti degli altri partecipanti all'ecosistema Linux

possono trarre beneficio attraverso il contributo al kernel. Inserire codice nel ramo principale è la chiave per arrivare ad una qualità del codice più alta, bassa manutenzione e bassi prezzi di distribuzione, alti livelli d'influenza sulla direzione dello sviluppo del kernel, e molto altro. È una situazione nella quale tutti coloro che sono coinvolti vincono. Mollate il vostro editor e raggiungeteci; sarete più che benvenuti.

Lo scopo di questo documento è quello di aiutare gli sviluppatori (ed i loro supervisori) a lavorare con la comunità di sviluppo con il minimo sforzo. È un tentativo di documentare il funzionamento di questa comunità in modo che sia accessibile anche a coloro che non hanno familiarità con lo sviluppo del Kernel Linux (o, anzi, con lo sviluppo di software libero in generale). Benchè qui sia presente del materiale tecnico, questa è una discussione rivolta in particolare al procedimento, e quindi per essere compreso non richiede una conoscenza approfondita sullo sviluppo del kernel.

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/submitting-patches.rst

**Translator** Federico Vaga <[federico.vaga@vaga.pv.it](mailto:federico.vaga@vaga.pv.it)>

### Inviare patch: la guida essenziale per vedere il vostro codice nel kernel

Una persona o un'azienda che volesse inviare una patch al kernel potrebbe sentirsi scoraggiata dal processo di sottomissione, specialmente quando manca una certa familiarità col "sistema". Questo testo è una raccolta di suggerimenti che aumenteranno significativamente le probabilità di vedere le vostre patch accettate.

Questo documento contiene un vasto numero di suggerimenti concisi. Per maggiori dettagli su come funziona il processo di sviluppo del kernel leggete [Una guida al processo di sviluppo del Kernel](#). Leggete anche [Lista delle verifiche da fare prima di inviare una patch per il kernel Linux](#) per una lista di punti da verificare prima di inviare del codice. Se state inviando un driver, allora leggete anche [Sottomettere driver per il kernel Linux](#); per delle patch relative alle associazioni per Device Tree leggete [Inviare patch: la guida essenziale per vedere il vostro codice nel kernel](#).

Questa documentazione assume che sappiate usare git per preparare le patch. Se non siete pratici di git, allora è bene che lo impariate; renderà la vostra vita di sviluppatore del kernel molto più semplice.

### Ottenere i sorgenti attuali

Se non avete un repository coi sorgenti del kernel più recenti, allora usate git per ottenerli. Vorrete iniziare col repository principale che può essere recuperato col comando:

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

Notate, comunque, che potreste non voler sviluppare direttamente coi sorgenti principali del kernel. La maggior parte dei manutentori hanno i propri sorgenti e desiderano che le patch siano preparate basandosi su di essi. Guardate l'elemento **T:** per un determinato sottosistema

nel file MAINTANERS che troverete nei sorgenti, o semplicemente chiedete al manutentore nel caso in cui i sorgenti da usare non siano elencati il quel file.

## Descrivete le vostre modifiche

Descrivete il vostro problema. Esiste sempre un problema che via ha spinto ha fare il vostro lavoro, che sia la correzione di un baco da una riga o una nuova funzionalità da 5000 righe di codice. Convincete i revisori che vale la pena risolvere il vostro problema e che ha senso continuare a leggere oltre al primo paragrafo.

Descrivete ciò che sarà visibile agli utenti. Chiari incidenti nel sistema e blocchi sono abbastanza convincenti, ma non tutti i bachi sono così evidenti. Anche se il problema è stato scoperto durante la revisione del codice, descrivete l'impatto che questo avrà sugli utenti. Tenete presente che la maggior parte delle installazioni Linux usa un kernel che arriva dai sorgenti stabili o dai sorgenti di una distribuzione particolare che prende singolarmente le patch dai sorgenti principali; quindi, includete tutte le informazioni che possono essere utili a capire le vostre modifiche: le circostanze che causano il problema, estratti da dmesg, descrizioni di un incidente di sistema, prestazioni di una regressione, picchi di latenza, blocchi, eccetera.

Quantificare le ottimizzazioni e i compromessi. Se affermate di aver migliorato le prestazioni, il consumo di memoria, l'impatto sullo stack, o la dimensione del file binario, includete dei numeri a supporto della vostra dichiarazione. Ma ricordatevi di descrivere anche eventuali costi che non sono ovvi. Solitamente le ottimizzazioni non sono gratuite, ma sono un compromesso fra l'uso di CPU, la memoria e la leggibilità; o, quando si parla di ipotesi euristiche, fra differenti carichi. Descrivete i lati negativi che vi aspettate dall'ottimizzazione cosicché i revisori possano valutare i costi e i benefici.

Una volta che il problema è chiaro, descrivete come lo risolvete andando nel dettaglio tecnico. È molto importante che descriviate la modifica in un inglese semplice cosicché i revisori possano verificare che il codice si comporti come descritto.

I manutentori vi saranno grati se scrivete la descrizione della patch in un formato che sia compatibile con il gestore dei sorgenti usato dal kernel, git, come un “commit log”. Leggete [Usare esplicitamente In-Reply-To nell'intestazione](#).

Risolvete solo un problema per patch. Se la vostra descrizione inizia ad essere lunga, potrebbe essere un segno che la vostra patch necessita d'essere divisa. Leggete [split\\_changes](#).

Quando inviate o rinviate una patch o una serie, includete la descrizione completa delle modifiche e la loro giustificazione. Non limitatevi a dire che questa è la versione N della patch (o serie). Non aspettatevi che i manutentori di un sottosistema vadano a cercare le versioni precedenti per cercare la descrizione da aggiungere. In pratica, la patch (o serie) e la sua descrizione devono essere un'unica cosa. Questo aiuta i manutentori e i revisori. Probabilmente, alcuni revisori non hanno nemmeno ricevuto o visto le versioni precedenti della patch.

Descrivete le vostre modifiche usando l'imperativo, per esempio “make xyzzy do frotz” piuttosto che “[This patch] makes xyzzy do frotz” or “[I] changed xyzzy to do frotz”, come se steste dando ordini al codice di cambiare il suo comportamento.

Se la patch corregge un baco conosciuto, fare riferimento a quel baco inserendo il suo numero o il suo URL. Se la patch è la conseguenza di una discussione su una lista di discussione, allora fornite l'URL all'archivio di quella discussione; usate i collegamenti a <https://lore.kernel.org/> con il Message-ID, in questo modo vi assicurerete che il collegamento non diventi invalido nel tempo.

Tuttavia, cercate di rendere la vostra spiegazione comprensibile anche senza far riferimento a fonti esterne. In aggiunta ai collegamenti a bachi e liste di discussione, riassumete i punti più importanti della discussione che hanno portato alla creazione della patch.

Se volete far riferimento a uno specifico commit, non usate solo l'identificativo SHA-1. Per cortesia, aggiungete anche la breve riga riassuntiva del commit per rendere la chiaro ai revisori l'oggetto. Per esempio:

```
Commit e21d2170f36602ae2708 ("video: remove unnecessary  
platform_set_drvdata()") removed the unnecessary  
platform_set_drvdata(), but left the variable "dev" unused,  
delete it.
```

Dovreste anche assicurarvi di usare almeno i primi 12 caratteri dell'identificativo SHA-1. Il repository del kernel ha *molti* oggetti e questo rende possibile la collisione fra due identificativi con pochi caratteri. Tenete ben presente che anche se oggi non ci sono collisioni con il vostro identificativo a 6 caratteri, potrebbero essercene fra 5 anni da oggi.

Se la vostra patch corregge un baco in un commit specifico, per esempio avete trovato un problema usando `git bisect`, per favore usate l'etichetta 'Fixes:' indicando i primi 12 caratteri dell'identificativo SHA-1 seguiti dalla riga riassuntiva. Per esempio:

```
Fixes: e21d2170f366 ("video: remove unnecessary platform_set_drvdata()")
```

La seguente configurazione di `git config` può essere usata per formattare i risultati dei comandi `git log` o `git show` come nell'esempio precedente:

```
[core]  
    abbrev = 12  
[pretty]  
    fixes = Fixes: %h (%s)
```

Un esempio:

```
$ git log -1 --pretty=fixes 54a4f0239f2e  
Fixes: 54a4f0239f2e ("KVM: MMU: make kvm_mmu_zap_page() return the number of  
→ pages it actually freed")
```

## Separate le vostre modifiche

Separate ogni **cambiamento logico** in patch distinte.

Per esempio, se i vostri cambiamenti per un singolo driver includono sia delle correzioni di bachi che miglioramenti alle prestazioni, allora separateli in due o più patch. Se i vostri cambiamenti includono un aggiornamento dell'API e un nuovo driver che lo sfrutta, allora separateli in due patch.

D'altro canto, se fate una singola modifica su più file, raggruppate tutte queste modifiche in una singola patch. Dunque, un singolo cambiamento logico è contenuto in una sola patch.

Il punto da ricordare è che ogni modifica dovrebbe fare delle modifiche che siano facilmente comprensibili e che possano essere verificate dai revisori. Ogni patch dovrebbe essere giustificabile di per sé.

Se al fine di ottenere un cambiamento completo una patch dipende da un'altra, va bene. Semplicemente scrivete una nota nella descrizione della patch per farlo presente: “**this patch depends on patch X**”.

Quando dividete i vostri cambiamenti in una serie di patch, prestate particolare attenzione alla verifica di ogni patch della serie; per ognuna il kernel deve compilare ed essere eseguito correttamente. Gli sviluppatori che usano `git bisect` per scovare i problemi potrebbero finire nel mezzo della vostra serie in un punto qualsiasi; non vi saranno grati se nel mezzo avete introdotto dei bachi.

Se non potete condensare la vostra serie di patch in una più piccola, allora pubblicatene una quindicina alla volta e aspettate che vengano revisionate ed integrate.

#### 4) Verify the style of your changes

Controllate che la vostra patch non violi lo stile del codice, maggiori dettagli sono disponibili in [Stile del codice per il kernel Linux](#). Non farlo porta semplicemente a una perdita di tempo da parte dei revisori e voi vedrete la vostra patch rifiutata, probabilmente senza nemmeno essere stata letta.

Un’eccezione importante si ha quando del codice viene spostato da un file ad un altro – in questo caso non dovreste modificare il codice spostato per nessun motivo, almeno non nella patch che lo sposta. Questo separa chiaramente l’azione di spostare il codice e il vostro cambiamento. Questo aiuta enormemente la revisione delle vere differenze e permette agli strumenti di tenere meglio la traccia della storia del codice.

Prima di inviare una patch, verificatene lo stile usando l’apposito verificatore (`scripts/checkpatch.pl`). Da notare, comunque, che il verificatore di stile dovrebbe essere visto come una guida, non come un sostituto al giudizio umano. Se il vostro codice è migliore nonostante una violazione dello stile, probabilmente è meglio lasciarlo com’è.

**The verifier has three levels of severity:**

- ERROR: le cose sono molto probabilmente sbagliate
- WARNING: le cose necessitano d’essere revisionate con attenzione
- CHECK: le cose necessitano di un pensierino

Dovreste essere in grado di giustificare tutte le eventuali violazioni rimaste nella vostra patch.

#### 5) Select the recipients of your patch

Dovreste sempre inviare una copia della patch ai manutentori dei sottosistemi interessati dalle modifiche; date un’occhiata al file `MAINTAINERS` e alla storia delle revisioni per scoprire chi si occupa del codice. Lo script `scripts/get_maintainer.pl` può esservi d’aiuto. Se non riuscite a trovare un manutentore per il sottosistema su cui state lavorando, allora Andrew Morton ([akpm@linux-foundation.org](mailto:akpm@linux-foundation.org)) sarà la vostra ultima possibilità.

Normalmente, dovreste anche scegliere una lista di discussione a cui inviare la vostra serie di patch. La lista di discussione [linux-kernel@vger.kernel.org](mailto:linux-kernel@vger.kernel.org) dovrebbe essere usata per inviare tutte le patch, ma il traffico è tale per cui diversi sviluppatori la trascurano. Guardate nel file `MAINTAINERS` per trovare la lista di discussione dedicata ad un sottosistema; probabilmente

lì la vostra patch riceverà molta più attenzione. Tuttavia, per favore, non spammate le liste di discussione che non sono interessate al vostro lavoro.

Molte delle liste di discussione relative al kernel vengono ospitate su vger.kernel.org; potete trovare un loro elenco alla pagina <http://vger.kernel.org/vger-lists.html>. Tuttavia, ci sono altre liste di discussione ospitate altrove.

Non inviate più di 15 patch alla volta sulle liste di discussione vger!!!

L'ultimo giudizio sull'integrazione delle modifiche accettate spetta a Linux Torvalds. Il suo indirizzo e-mail è <[torvalds@linux-foundation.org](mailto:torvalds@linux-foundation.org)>. Riceve moltissime e-mail, e, a questo punto, solo poche patch passano direttamente attraverso il suo giudizio; quindi, dovreste fare del vostro meglio per -evitare di- inviar gli e-mail.

Se avete una patch che corregge un baco di sicurezza che potrebbe essere sfruttato, inviatela a [security@kernel.org](mailto:security@kernel.org). Per bachi importanti, un breve embargo potrebbe essere preso in considerazione per dare il tempo alle distribuzioni di prendere la patch e renderla disponibile ai loro utenti; in questo caso, ovviamente, la patch non dovrebbe essere inviata su alcuna lista di discussione pubblica. Leggete anche Documentation/admin-guide/security-bugs.rst.

Patch che correggono bachi importanti su un kernel già rilasciato, dovrebbero essere inviate ai manutentori dei kernel stabili aggiungendo la seguente riga:

Cc: [stable@vger.kernel.org](mailto:stable@vger.kernel.org)

nella vostra patch, nell'area dedicata alle firme (notate, NON come destinatario delle e-mail). In aggiunta a questo file, dovreste leggere anche *Tutto quello che volevate sapere sui rilasci -stable di Linux*.

Se le modifiche hanno effetti sull'interfaccia con lo spazio utente, per favore inviate una patch per le pagine man ai manutentori di sudette pagine (elencati nel file MAINTAINERS), o almeno una notifica circa la vostra modifica, cosicché l'informazione possa trovare la sua strada nel manuale. Le modifiche all'API dello spazio utente dovrebbero essere inviate in copia anche a [linux-api@vger.kernel.org](mailto:linux-api@vger.kernel.org).

### Niente: MIME, links, compressione, allegati. Solo puro testo

Linus e gli altri sviluppatori del kernel devono poter commentare le modifiche che sottomettete. Per uno sviluppatore è importante essere in grado di "citare" le vostre modifiche, usando normali programmi di posta elettronica, cosicché sia possibile commentare una porzione specifica del vostro codice.

Per questa ragione tutte le patch devono essere inviate via e-mail come testo. Il modo più facile, e quello raccomandato, è con `git send-email`. Al sito <https://git-send-email.io> è disponibile una guida interattiva sull'uso di `git send-email`.

Se decidete di non usare `git send-email`:

**Warning:** Se decidete di copiare ed incollare la patch nel corpo dell'e-mail, state attenti che il vostro programma non corrompa il contenuto con andate a capo automatiche.

La patch non deve essere un allegato MIME, compresso o meno. Molti dei più popolari programmi di posta elettronica non trasmettono un allegato MIME come puro testo, e questo rende

impossibile commentare il vostro codice. Inoltre, un allegato MIME rende l'attività di Linus più laboriosa, diminuendo così la possibilità che il vostro allegato-MIME venga accettato.

Eccezione: se il vostro servizio di posta storpia le patch, allora qualcuno potrebbe chiedervi di rinviarle come allegato MIME.

Leggete *Informazioni sui programmi di posta elettronica per Linux* per dei suggerimenti sulla configurazione dei programmi di posta elettronica per l'invio di patch intatte.

## Rispondere ai commenti di revisione

In risposta alla vostra email, quasi certamente i revisori vi invieranno dei commenti su come migliorare la vostra patch. Dovete rispondere a questi commenti; ignorare i revisori è un ottimo modo per essere ignorati. Riscontri o domande che non conducono ad una modifica del codice quasi certamente dovrebbero portare ad un commento nel changelog cosicché il prossimo revisore potrà meglio comprendere cosa stia accadendo.

Assicuratevi di dire ai revisori quali cambiamenti state facendo e di ringraziarli per il loro tempo. Revisionare codice è un lavoro faticoso e che richiede molto tempo, e a volte i revisori diventano burberi. Tuttavia, anche in questo caso, rispondete con educazione e concentratevi sul problema che hanno evidenziato.

Leggete *Informazioni sui programmi di posta elettronica per Linux* per le raccomandazioni sui programmi di posta elettronica e l'etichetta da usare sulle liste di discussione.

## Non scoraggiatevi - o impazientitevi

Dopo che avete inviato le vostre modifiche, state pazienti e aspettate. I revisori sono persone occupate e potrebbero non ricevere la vostra patch immediatamente.

Un tempo, le patch erano solite scomparire nel vuoto senza alcun commento, ma ora il processo di sviluppo funziona meglio. Dovreste ricevere commenti in una settimana o poco più; se questo non dovesse accadere, assicuratevi di aver inviato le patch correttamente. Aspettate almeno una settimana prima di rinviare le modifiche o sollecitare i revisori - probabilmente anche di più durante la finestra d'integrazione.

Potete anche rinviare la patch, o la serie di patch, dopo un paio di settimane aggiungendo la parola "RESEND" nel titolo:

```
[PATCH Vx RESEND] sub/sys: Condensed patch summary
```

Ma non aggiungete "RESEND" quando state sottomettendo una versione modificata della vostra patch, o serie di patch - "RESEND" si applica solo alla sottomissione di patch, o serie di patch, che non hanno subito modifiche dall'ultima volta che sono state inviate.

### Aggiungete PATCH nell'oggetto

Dato l'alto volume di e-mail per Linus, e la lista linux-kernel, è prassi prefiggere il vostro oggetto con [PATCH]. Questo permette a Linus e agli altri sviluppatori del kernel di distinguere facilmente le patch dalle altre discussioni.

`git send-email` lo fa automaticamente.

### Firmate il vostro lavoro - Il certificato d'origine dello sviluppatore

Per migliorare la tracciabilità su “chi ha fatto cosa”, specialmente per quelle patch che per raggiungere lo stadio finale passano attraverso diversi livelli di manutentori, abbiamo introdotto la procedura di “firma” delle patch che vengono inviate per e-mail.

La firma è una semplice riga alla fine della descrizione della patch che certifica che l'avete scritta voi o che avete il diritto di pubblicarla come patch open-source. Le regole sono abbastanza semplici: se potete certificare quanto segue:

#### Il certificato d'origine dello sviluppatore 1.1

Contribuendo a questo progetto, io certifico che:

- (a) Il contributo è stato creato interamente, o in parte, da me e che ho il diritto di inviarlo in accordo con la licenza open-source indicata nel file; oppure
- (b) Il contributo è basato su un lavoro precedente che, nei limiti della mia conoscenza, è coperto da un'appropriata licenza open-source che mi da il diritto di modificarlo e inviarlo, le cui modifiche sono interamente o in parte mie, in accordo con la licenza open-source (a meno che non abbia il permesso di usare un'altra licenza) indicata nel file; oppure
- (c) Il contributo mi è stato fornito direttamente da qualcuno che ha certificato (a), (b) o (c) e non l'ho modificata.
- (d) Capisco e concordo col fatto che questo progetto e i suoi contributi sono pubblici e che un registro dei contributi (incluse tutte le informazioni personali che invio con essi, inclusa la mia firma) verrà mantenuto indefinitamente e che possa essere ridistribuito in accordo con questo progetto o le licenze open-source coinvolte.

poi dovete solo aggiungere una riga che dice:

```
Signed-off-by: Random J Developer <random@developer.example.org>
```

usando il vostro vero nome (spiacenti, non si accettano pseudonimi o contributi anonimi). Questo verrà fatto automaticamente se usate `git commit -s`. Anche il ripristino di uno stato precedente dovrebbe includere “Signed-off-by”, se usate `git revert -s` questo verrà fatto automaticamente.

Alcune persone aggiungono delle etichette alla fine. Per ora queste verranno ignorate, ma potete farlo per meglio identificare procedure aziendali interne o per aggiungere dettagli circa la firma.

In seguito al SoB (Signed-off-by:) dell'autore ve ne sono altri da parte di tutte quelle persone che si sono occupate della gestione e del trasporto della patch. Queste però non sono state

coinvolte nello sviluppo, ma la loro sequenza d'apparizione ci racconta il percorso **reale** che una patch a intrapreso dallo sviluppatore, fino al manutentore, per poi giungere a Linus.

## Quando utilizzare Acked-by:, Cc:, e Co-developed-by:

L'etichetta Signed-off-by: indica che il firmatario è stato coinvolto nello sviluppo della patch, o che era nel suo percorso di consegna.

Se una persona non è direttamente coinvolta con la preparazione o gestione della patch ma desidera firmare e mettere agli atti la loro approvazione, allora queste persone possono chiedere di aggiungere al changelog della patch una riga Acked-by:.

Acked-by: viene spesso utilizzato dai manutentori del sottosistema in oggetto quando quello stesso manutentore non ha contribuito né trasmesso la patch.

Acked-by: non è formale come Signed-off-by:. Questo indica che la persona ha revisionato la patch e l'ha trovata accettabile. Per cui, a volte, chi integra le patch convertirà un "sì, mi sembra che vada bene" in un Acked-by: (ma tenete presente che solitamente è meglio chiedere esplicitamente).

Acked-by: non indica l'accettazione di un'intera patch. Per esempio, quando una patch ha effetti su diversi sottosistemi e ha un Acked-by: da un manutentore di uno di questi, significa che il manutentore accetta quella parte di codice relativa al sottosistema che mantiene. Qui dovremmo essere giudiziari. Quando si hanno dei dubbi si dovrebbe far riferimento alla discussione originale negli archivi della lista di discussione.

Se una persona ha avuto l'opportunità di commentare la patch, ma non lo ha fatto, potete aggiungere l'etichetta Cc: alla patch. Questa è l'unica etichetta che può essere aggiunta senza che la persona in questione faccia alcunché - ma dovrebbe indicare che la persona ha ricevuto una copia della patch. Questa etichetta documenta che terzi potenzialmente interessati sono stati inclusi nella discussione.

Co-developed-by: indica che la patch è stata cosviluppata da diversi sviluppatori; viene usato per assegnare più autori (in aggiunta a quello associato all'etichetta From:) quando più persone lavorano ad una patch. Dato che Co-developed-by: implica la paternità della patch, ogni Co-developed-by: dev'essere seguito immediatamente dal Signed-off-by: del corrispondente coautore. Qui si applica la procedura di base per sign-off, in pratica l'ordine delle etichette Signed-off-by: dovrebbe riflettere il più possibile l'ordine cronologico della storia della patch, indipendentemente dal fatto che la paternità venga assegnata via From: o Co-developed-by:. Da notare che l'ultimo Signed-off-by: dev'essere quello di colui che ha sottomesso la patch.

Notate anche che l'etichetta From: è opzionale quando l'autore in From: è anche la persona (e indirizzo email) indicato nel From: dell'intestazione dell'email.

Esempio di una patch sottomessa dall'autore in From::

```
<changelog>

Co-developed-by: First Co-Author <first@coauthor.example.org>
Signed-off-by: First Co-Author <first@coauthor.example.org>
Co-developed-by: Second Co-Author <second@coauthor.example.org>
Signed-off-by: Second Co-Author <second@coauthor.example.org>
Signed-off-by: From Author <from@author.example.org>
```

Esempio di una patch sottomessa dall'autore Co-developed-by::

```
From: From Author <from@author.example.org>
<changelog>

Co-developed-by: Random Co-Author <random@coauthor.example.org>
Signed-off-by: Random Co-Author <random@coauthor.example.org>
Signed-off-by: From Author <from@author.example.org>
Co-developed-by: Submitting Co-Author <sub@coauthor.example.org>
Signed-off-by: Submitting Co-Author <sub@coauthor.example.org>
```

### Utilizzare Reported-by:, Tested-by:, Reviewed-by:, Suggested-by: e Fixes:

L'etichetta Reported-by da credito alle persone che trovano e riportano i bachi e si spera che questo possa ispirarli ad aiutarci nuovamente in futuro. Rammentate che se il baco è stato riportato in privato, dovrete chiedere il permesso prima di poter utilizzare l'etichetta Reported-by.

L'etichetta Tested-by: indica che la patch è stata verificata con successo (su un qualche sistema) dalla persona citata. Questa etichetta informa i manutentori che qualche verifica è stata fatta, fornisce un mezzo per trovare persone che possano verificare il codice in futuro, e garantisce che queste stesse persone ricevano credito per il loro lavoro.

Reviewd-by:, invece, indica che la patch è stata revisionata ed è stata considerata accettabile in accordo con la dichiarazione dei revisori:

### Dichiarazione di svista dei revisori

Offrendo la mia etichetta Reviewed-by, dichiaro quanto segue:

- Ho effettuato una revisione tecnica di questa patch per valutarne l'adeguatezza ai fini dell'inclusione nel ramo principale del kernel.
- Tutti i problemi e le domande riguardanti la patch sono stati comunicati al mittente. Sono soddisfatto dalle risposte del mittente.
- Nonostante ci potrebbero essere cose migliorabili in queste sottomissioni, credo che sia, in questo momento, (1) una modifica di interesse per il kernel, e (2) libera da problemi che potrebbero metterne in discussione l'integrazione.
- Nonostante abbia revisionato la patch e creda che vada bene, non garantisco (se non specificato altrimenti) che questa otterrà quello che promette o funzionerà correttamente in tutte le possibili situazioni.

L'etichetta Reviewed-by è la dichiarazione di un parere sulla bontà di una modifica che si ritiene appropriata e senza alcun problema tecnico importante. Qualsiasi revisore interessato (quelli che lo hanno fatto) possono offrire il proprio Reviewed-by per la patch. Questa etichetta serve a dare credito ai revisori e a informare i manutentori sul livello di revisione che è stato fatto sulla patch. L'etichetta Reviewd-by, quando fornita da revisori conosciuti per la loro conoscenza sulla materia in oggetto e per la loro serietà nella revisione, accrescerà le probabilità che la vostra patch venga integrate nel kernel.

Quando si riceve una email sulla lista di discussione da un tester o un revisore, le etichette Tested-by o Reviewd-by devono essere aggiunte dall'autore quando invierà nuovamente la patch. Tuttavia, se la patch è cambiata in modo significativo, queste etichette potrebbero non avere più senso e quindi andrebbero rimosse. Solitamente si tiene traccia della rimozione nel changelog della patch (subito dopo il separatore ‘—’).

L'etichetta Suggested-by: indica che l'idea della patch è stata suggerita dalla persona nominata e le da credito. Tenete a mente che questa etichetta non dovrebbe essere aggiunta senza un permesso esplicito, specialmente se l'idea non è stata pubblicata in un forum pubblico. Detto ciò, dando credito a chi ci fornisce delle idee, si spera di poterli ispirare ad aiutarci nuovamente in futuro.

L'etichetta Fixes: indica che la patch corregge un problema in un commit precedente. Serve a chiarire l'origine di un baco, il che aiuta la revisione del baco stesso. Questa etichetta è di aiuto anche per i manutentori dei kernel stabili al fine di capire quale kernel deve ricevere la correzione. Questo è il modo suggerito per indicare che un baco è stato corretto nella patch. Per maggiori dettagli leggete [Descrivete le vostre modifiche](#)

Da notare che aggiungere un tag “Fixes:” non esime dalle regole previste per i kernel stabili, e nemmeno dalla necessità di aggiungere in copia conoscenza [stable@vger.kernel.org](mailto:stable@vger.kernel.org) su tutte le patch per suddetti kernel.

## Il formato canonico delle patch

Questa sezione descrive il formato che dovrebbe essere usato per le patch. Notate che se state usando un repository git per salvare le vostre patch potrete usare il comando `git format-patch` per ottenere patch nel formato appropriato. Lo strumento non crea il testo necessario, per cui, leggete le seguenti istruzioni.

L'oggetto di una patch canonica è la riga:

Subject: [PATCH 001/123] subsystem: summary phrase
--

Il corpo di una patch canonica contiene i seguenti elementi:

- Una riga `from` che specifica l'autore della patch, seguita da una riga vuota (necessaria soltanto se la persona che invia la patch non ne è l'autore).
- Il corpo della spiegazione, con linee non più lunghe di 75 caratteri, che verrà copiato permanentemente nel changelog per descrivere la patch.
- Una riga vuota
- Le righe `Signed-off-by:`, descritte in precedenza, che finiranno anch'esse nel changelog.
- Una linea di demarcazione contenente semplicemente `---`.
- Qualsiasi altro commento che non deve finire nel changelog.
- Le effettive modifiche al codice (il prodotto di `diff`).

Il formato usato per l'oggetto permette ai programmi di posta di usarlo per ordinare le patch alfabeticamente - tutti i programmi di posta hanno questa funzionalità - dato che al numero sequenziale si antepongono degli zeri; in questo modo l'ordine numerico ed alfabetico coincidono.

Il `subsystem` nell'oggetto dell'email dovrebbe identificare l'area o il sottosistema modificato dalla patch.

La **summary phrase** nell'oggetto dell'email dovrebbe descrivere brevemente il contenuto della patch. La **summary phrase** non dovrebbe essere un nome di file. Non utilizzate la stessa **summary phrase** per tutte le patch in una serie (dove una **serie di patch** è una sequenza ordinata di diverse patch correlate).

Ricordatevi che la **summary phrase** della vostra email diventerà un identificatore globale ed unico per quella patch. Si propaga fino al changelog git. La **summary phrase** potrà essere usata in futuro dagli sviluppatori per riferirsi a quella patch. Le persone vorranno cercare la **summary phrase** su internet per leggere le discussioni che la riguardano. Potrebbe anche essere l'unica cosa che le persone vedranno quando, in due o tre mesi, riguarderanno centinaia di patch usando strumenti come `gitk` o `git log --oneline`.

Per queste ragioni, dovrebbe essere lunga fra i 70 e i 75 caratteri, e deve descrivere sia cosa viene modificato, sia il perché sia necessario. Essere brevi e descrittivi è una bella sfida, ma questo è quello che fa un riassunto ben scritto.

La **summary phrase** può avere un'etichetta (*tag*) di prefisso racchiusa fra le parentesi quadre “`Subject: [PATCH <tag>...] <summary phrase>`”. Le etichette non verranno considerate come parte della frase riassuntiva, ma indicano come la patch dovrebbe essere trattata. Fra le etichette più comuni ci sono quelle di versione che vengono usate quando una patch è stata inviata più volte (per esempio, “v1, v2, v3”); oppure “RFC” per indicare che si attendono dei commenti (*Request For Comments*).

Se ci sono quattro patch nella serie, queste dovrebbero essere enumerate così: 1/4, 2/4, 3/4, 4/4. Questo assicura che gli sviluppatori capiranno l'ordine in cui le patch dovrebbero essere applicate, e per tracciare quelle che hanno revisionate o che hanno applicato.

Un paio di esempi di oggetti:

```
Subject: [PATCH 2/5] ext2: improve scalability of bitmap searching
Subject: [PATCH v2 01/27] x86: fix eflags tracking
Subject: [PATCH v2] sub/sys: Condensed patch summary
Subject: [PATCH v2 M/N] sub/sys: Condensed patch summary
```

La riga `from` dev'essere la prima nel corpo del messaggio ed è nel formato:

From: Patch Author <[author@example.com](mailto:author@example.com)>

La riga `from` indica chi verrà accreditato nel changelog permanente come l'autore della patch. Se la riga `from` è mancante, allora per determinare l'autore da inserire nel changelog verrà usata la riga `From` nell'intestazione dell'email.

Il corpo della spiegazione verrà incluso nel changelog permanente, per cui deve aver senso per un lettore esperto che è ha dimenticato i dettagli della discussione che hanno portato alla patch. L'inclusione di informazioni sui problemi oggetto dalla patch (messaggi del kernel, messaggi di oops, eccetera) è particolarmente utile per le persone che potrebbero cercare fra i messaggi di log per la patch che li tratta. Il testo dovrebbe essere scritto con abbastanza dettagli da far capire al lettore **perché** quella patch fu creata, e questo a distanza di settimane, mesi, o addirittura anni.

Se la patch corregge un errore di compilazione, non sarà necessario includere proprio tutto quello che è uscito dal compilatore; aggiungete solo quello che è necessario per far sì che la vostra patch venga trovata. Come nella **summary phrase**, è importante essere sia brevi che descrittivi.

La linea di demarcazione `---` serve essenzialmente a segnare dove finisce il messaggio di

changelog.

Aggiungere il `diffstat` dopo `---` è un buon uso di questo spazio, per mostrare i file che sono cambiati, e il numero di file aggiunto o rimossi. Un `diffstat` è particolarmente utile per le patch grandi. Se includete un `diffstat` dopo `---`, usate le opzioni `-p 1 -w70` cosicché i nomi dei file elencati non occupino troppo spazio (facilmente rientrano negli 80 caratteri, magari con qualche indentazione). (`git` genera di base dei `diffstat` adatti).

I commenti che sono importanti solo per i manutentori, quindi inadatti al changelog permanente, dovrebbero essere messi qui. Un buon esempio per questo tipo di commenti potrebbe essere il cosiddetto `patch changelogs` che descrivere le differenze fra le versioni della patch.

Queste informazioni devono andare **dopo** la linea `---` che separa il *changelog* dal resto della patch. Le informazioni riguardanti la versione di una patch non sono parte del *changelog* che viene incluso in `git`. Queste sono informazioni utili solo ai revisori. Se venissero messe sopra la riga, qualcuno dovrà fare del lavoro manuale per rimuoverle; cosa che invece viene fatta automaticamente quando vengono messe correttamente oltre la riga.:

```
<commit message>
...
Signed-off-by: Author <author@mail>
---
V2 -> V3: Removed redundant helper function
V1 -> V2: Cleaned up coding style and addressed review comments

path/to/file | 5+---+
...
```

Maggiori dettagli sul formato delle patch nei riferimenti qui di seguito.

## Aggiungere i *backtrace* nei messaggi di commit

I *backtrace* aiutano a documentare la sequenza di chiamate a funzione che portano ad un problema. Tuttavia, non tutti i *backtrace* sono davvero utili. Per esempio, le sequenze iniziali di avvio sono uniche e ovvie. Copiare integralmente l'output di `dmesg` aggiunge tante informazioni che distraggono dal vero problema (per esempio, i marcatori temporali, la lista dei moduli, la lista dei registri, lo stato dello stack).

Quindi, per rendere utile un *backtrace* dovreste eliminare le informazioni inutili, cosicché ci si possa focalizzare sul problema. Ecco un esempio di un *backtrace* essenziale:

```
unchecked MSR access error: WRMSR to 0xd51 (tried to write 0x0000000000000064)
at rIP: 0xfffffffffae059994 (native_write_msr+0x4/0x20)
Call Trace:
mba_wrmsr
update_domains
rdtgroup_mkdir
```

### Usare esplicitamente In-Reply-To nell'intestazione

Aggiungere manualmente In-Reply-To: nell'intestazione dell'e-mail potrebbe essere d'aiuto per associare una patch ad una discussione precedente, per esempio per collegare la correzione di un baco con l'e-mail che lo riportava. Tuttavia, per serie di patch multiple è generalmente sconsigliato l'uso di In-Reply-To: per collegare precedenti versioni. In questo modo versioni multiple di una patch non diventeranno un'ingestibile giungla di riferimenti all'interno dei programmi di posta. Se un collegamento è utile, potete usare <https://lore.kernel.org/> per ottenere i collegamenti ad una versione precedente di una serie di patch (per esempio, potete usarlo per l'email introduttiva alla serie).

### Riferimenti

**Andrew Morton, "La patch perfetta" (tpp).** <<http://www.ozlabs.org/~akpm/stuff/tpp.txt>>

**Jeff Garzik, "Formato per la sottomissione di patch per il kernel Linux"** <<https://web.archive.org/web/20180829112450/http://linux.yyz.us/patch-format.html>>

**Greg Kroah-Hartman, "Come scacciare un manutentore di un sottosistema"**

<<http://www.kroah.com/log/linux/maintainer.html>>

<<http://www.kroah.com/log/linux/maintainer-02.html>>

<<http://www.kroah.com/log/linux/maintainer-03.html>>

<<http://www.kroah.com/log/linux/maintainer-04.html>>

<<http://www.kroah.com/log/linux/maintainer-05.html>>

<<http://www.kroah.com/log/linux/maintainer-06.html>>

**No!!!! Basta gigantesche bombe patch alle persone sulla lista linux-kernel@vger.kernel.org**  
<<https://lore.kernel.org/r/20050711.125305.08322243.davem@davemloft.net>>

Kernel *Stile del codice per il kernel Linux.*

**E-mail di Linus Torvalds sul formato canonico di una patch:** <<https://lore.kernel.org/r/Pine.LNX.4.58.0504071023190.28951@ppc970.osdl.org>>

**Andi Kleen, "Su come sottomettere patch del kernel"** Alcune strategie su come sottomettere modifiche toste o controverse.

<http://halobates.de/on-submitting-patches.pdf>

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/programming-language.rst

**Translator** Federico Vaga <[federico.vaga@vaga.pv.it](mailto:federico.vaga@vaga.pv.it)>

## Linguaggio di programmazione

Il kernel è scritto nel linguaggio di programmazione C [[it-c-language](#)]. Più precisamente, il kernel viene compilato con gcc [[it-gcc](#)] usando l'opzione `-std=gnu11` [[it-gcc-c-dialect-options](#)]: il dialetto GNU dello standard ISO C11. Linux supporta anche clang [[it-clang](#)], leggete la documentazione Building Linux with Clang/LLVM.

Questo dialetto contiene diverse estensioni al linguaggio [[it-gnu-extensions](#)], e molte di queste vengono usate sistematicamente dal kernel.

Il kernel offre un certo livello di supporto per la compilazione con icc [[it-icc](#)] su diverse architetture, tuttavia in questo momento il supporto non è completo e richiede delle patch aggiuntive.

## Attributi

Una delle estensioni più comuni e usate nel kernel sono gli attributi [[it-gcc-attribute-syntax](#)]. Gli attributi permettono di aggiungere una semantica, definita dell'implementazione, alle entità del linguaggio (come le variabili, le funzioni o i tipi) senza dover fare importanti modifiche sintattiche al linguaggio stesso (come l'aggiunta di nuove parole chiave) [[it-n2049](#)].

In alcuni casi, gli attributi sono opzionali (ovvero un compilatore che non dovesse supportarli dovrebbe produrre comunque codice corretto, anche se più lento o che non esegue controlli aggiuntivi durante la compilazione).

Il kernel definisce alcune pseudo parole chiave (per esempio `__pure`) in alternativa alla sintassi GNU per gli attributi (per esempio `__attribute__((__pure__))`) allo scopo di mostrare quali funzionalità si possono usare e/o per accorciare il codice.

Per maggiori informazioni consultate il file d'intestazione `include/linux/compiler_attributes.h`.

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/coding-style.rst

**Translator** Federico Vaga <[federico.vaga@vaga.pv.it](mailto:federico.vaga@vaga.pv.it)>

## Stile del codice per il kernel Linux

Questo è un breve documento che descrive lo stile di codice preferito per il kernel Linux. Lo stile di codifica è molto personale e non voglio **forzare** nessuno ad accettare il mio, ma questo stile è quello che dev'essere usato per qualsiasi cosa che io sia in grado di mantenere, e l'ho preferito anche per molte altre cose. Per favore, almeno tenete in considerazione le osservazioni espresse qui.

La prima cosa che suggerisco è quella di stamparsi una copia degli standard di codifica GNU e di NON leggerla. Bruciatela, è un grande gesto simbolico.

Comunque, ecco i punti:

### 1) Indentazione

La tabulazione (tab) è di 8 caratteri e così anche le indentazioni. Ci sono alcuni movimenti di eretici che vorrebbero l'indentazione a 4 (o perfino 2!) caratteri di profondità, che è simile al tentativo di definire il valore del pi-greco a 3.

Motivazione: l'idea dell'indentazione è di definire chiaramente dove un blocco di controllo inizia e finisce. Specialmente quando siete rimasti a guardare lo schermo per 20 ore a file, troverete molto più facile capire i livelli di indentazione se questi sono larghi.

Ora, alcuni rivendicano che un'indentazione da 8 caratteri sposta il codice troppo a destra e che quindi rende difficile la lettura su schermi a 80 caratteri. La risposta a questa affermazione è che se vi servono più di 3 livelli di indentazione, siete comunque fregati e dovreste correggere il vostro programma.

In breve, l'indentazione ad 8 caratteri rende più facile la lettura, e in aggiunta vi avvisa quando state annidando troppo le vostre funzioni. Tenete ben a mente questo avviso.

Al fine di facilitare l'indentazione del costrutto switch, si preferisce allineare sulla stessa colonna la parola chiave switch e i suoi subordinati case. In questo modo si evita una doppia indentazione per i case. Un esempio.:

```
switch (suffix) {  
    case 'G':  
    case 'g':  
        mem <= 30;  
        break;  
    case 'M':  
    case 'm':  
        mem <= 20;  
        break;  
    case 'K':  
    case 'k':  
        mem <= 10;  
        fallthrough;  
    default:  
        break;  
}
```

A meno che non vogliate nascondere qualcosa, non mettete più istruzioni sulla stessa riga:

```
if (condition) do_this;  
do_something_evertime;
```

Non usate le virgolette per evitare le parentesi:

```
if (condition)  
    do_this(), do_that();
```

Invece, usate sempre le parentesi per racchiudere più istruzioni.

```
if (condition) {  
    do_this();
```

```
    do_that();
}
```

Non mettete nemmeno più assegnamenti sulla stessa riga. Lo stile del kernel è ultrasemplice. Evitate espressioni intricate.

Al di fuori dei commenti, della documentazione ed escludendo i Kconfig, gli spazi non vengono mai usati per l'indentazione, e l'esempio qui sopra è volutamente errato.

Procuratevi un buon editor di testo e non lasciate spazi bianchi alla fine delle righe.

## 2) Spezzare righe lunghe e stringhe

Lo stile del codice riguarda la leggibilità e la manutenibilità utilizzando strumenti comuni.

Come limite di riga si preferiscono le 80 colonne.

Espressioni più lunghe di 80 colonne dovrebbero essere spezzettate in pezzi più piccoli, a meno che eccedere le 80 colonne non aiuti ad aumentare la leggibilità senza nascondere informazioni.

I nuovi pezzi derivati sono sostanzialmente più corti degli originali e vengono posizionati più a destra. Uno stile molto comune è quello di allineare i nuovi pezzi alla parentesi aperta di una funzione.

Lo stesso si applica, nei file d'intestazione, alle funzioni con una lista di argomenti molto lunga.

Tuttavia, non spezzettate mai le stringhe visibili agli utenti come i messaggi di printk, questo perché inibireste la possibilità d'utilizzare grep per cercarle.

## 3) Posizionamento di parentesi graffe e spazi

Un altro problema che s'affronta sempre quando si parla di stile in C è il posizionamento delle parentesi graffe. Al contrario della dimensione dell'indentazione, non ci sono motivi tecnici sulla base dei quali scegliere una strategia di posizionamento o un'altra; ma il modo qui preferito, come mostratoci dai profeti Kernighan e Ritchie, è quello di posizionare la parentesi graffa di apertura per ultima sulla riga, e quella di chiusura per prima su una nuova riga, così:

```
if (x is true) {
    we do y
}
```

Questo è valido per tutte le espressioni che non siano funzioni (if, switch, for, while, do). Per esempio:

```
switch (action) {
case KOBJ_ADD:
    return "add";
case KOBJ_REMOVE:
    return "remove";
case KOBJ_CHANGE:
    return "change";
default:
```

```
    return NULL;  
}
```

Tuttavia, c'è il caso speciale, le funzioni: queste hanno la parentesi graffa di apertura all'inizio della riga successiva, quindi:

```
int function(int x)  
{  
    body of function  
}
```

Eretici da tutto il mondo affermano che questa incoerenza è ... insomma ... incoerente, ma tutte le persone ragionevoli sanno che (a) K&R hanno **ragione** e (b) K&R hanno ragione. A parte questo, le funzioni sono comunque speciali (non potete annidarle in C).

Notate che la graffa di chiusura è da sola su una riga propria, ad **eccezione** di quei casi dove è seguita dalla continuazione della stessa espressione, in pratica `while` nell'espressione do-while, oppure `else` nell'espressione if-else, come questo:

```
do {  
    body of do-loop  
} while (condition);
```

e

```
if (x == y) {  
    ..  
} else if (x > y) {  
    ...  
} else {  
    ....  
}
```

Motivazione: K&R.

Inoltre, notate che questo posizionamento delle graffe minimizza il numero di righe vuote senza perdere di leggibilità. In questo modo, dato che le righe sul vostro schermo non sono una risorsa illimitata (pensate ad uno terminale con 25 righe), avrete delle righe vuote da riempire con dei commenti.

Non usate inutilmente le graffe dove una singola espressione è sufficiente.

```
if (condition)  
    action();
```

e

```
if (condition)  
    do_this();  
else  
    do_that();
```

Questo non vale nel caso in cui solo un ramo dell'espressione if-else contiene una sola espressione; in quest'ultimo caso usate le graffe per entrambe i rami:

```
if (condition) {
    do_this();
    do_that();
} else {
    otherwise();
}
```

Inoltre, usate le graffe se un ciclo contiene più di una semplice istruzione:

```
while (condition) {
    if (test)
        do_something();
}
```

### 3.1) Spazi

Lo stile del kernel Linux per quanto riguarda gli spazi, dipende (principalmente) dalle funzioni e dalle parole chiave. Usate uno spazio dopo (quasi tutte) le parole chiave. L'eccezione più evidente sono sizeof, typeof, alignof, e `_attribute_`, il cui aspetto è molto simile a quello delle funzioni (e in Linux, solitamente, sono usate con le parentesi, anche se il linguaggio non lo richiede; come `sizeof info` dopo aver dichiarato `struct fileinfo info`).

Quindi utilizzate uno spazio dopo le seguenti parole chiave:

```
if, switch, case, for, do, while
```

ma non con sizeof, typeof, alignof, o `_attribute_`. Ad esempio,

```
s = sizeof(struct file);
```

Non aggiungete spazi attorno (dentro) ad un'espressione fra parentesi. Questo esempio è **brutto**:

```
s = sizeof( struct file );
```

Quando dichiarate un puntatore ad una variabile o una funzione che ritorna un puntatore, il posto suggerito per l'asterisco \* è adiacente al nome della variabile o della funzione, e non adiacente al nome del tipo. Esempi:

```
char *linux_banner;
unsigned long long memparse(char *ptr, char **retptr);
char *match_strdup(substring_t *s);
```

Usate uno spazio attorno (da ogni parte) alla maggior parte degli operatori binari o ternari, come i seguenti:

```
= + - < > * / % | & ^ <= >= == != ? :
```

ma non mettete spazi dopo gli operatori unari:

```
& * + - ~ ! sizeof typeof alignof __attribute__ defined
```

nessuno spazio dopo l'operatore unario suffisso di incremento o decremento:

```
++ --
```

nessuno spazio dopo l'operatore unario prefisso di incremento o decremento:

```
++ --
```

e nessuno spazio attorno agli operatori dei membri di una struttura . e ->.

Non lasciate spazi bianchi alla fine delle righe. Alcuni editor con l'indentazione furba inseriranno gli spazi bianchi all'inizio di una nuova riga in modo appropriato, quindi potrete scrivere la riga di codice successiva immediatamente. Tuttavia, alcuni di questi stessi editor non rimuovono questi spazi bianchi quando non scrivete nulla sulla nuova riga, ad esempio perché volete lasciare una riga vuota. Il risultato è che finirete per avere delle righe che contengono spazi bianchi in coda.

Git vi avviserà delle modifiche che aggiungono questi spazi vuoti di fine riga, e può optionalmente rimuoverli per conto vostro; tuttavia, se state applicando una serie di modifiche, questo potrebbe far fallire delle modifiche successive perché il contesto delle righe verrà cambiato.

## 4) Assegnare nomi

C è un linguaggio spartano, e così dovrebbero esserlo i vostri nomi. Al contrario dei programmatori Modula-2 o Pascal, i programmatori C non usano nomi graziosi come ThisVariableIsATemporaryCounter. Un programmatore C chiamerebbe questa variabile `tmp`, che è molto più facile da scrivere e non è una delle più difficili da capire.

TUTTAVIA, nonostante i nomi con notazione mista siano da condannare, i nomi descrittivi per variabili globali sono un dovere. Chiamare una funzione globale `pioppo` è un insulto.

Le variabili GLOBALI (da usare solo se vi servono **davvero**) devono avere dei nomi descrittivi, così come le funzioni globali. Se avete una funzione che conta gli utenti attivi, dovreste chiamarla `count_active_users()` o qualcosa di simile, **non** dovreste chiamarla `cntusr()`.

Codificare il tipo di funzione nel suo nome (quella cosa chiamata notazione ungherese) è stupido - il compilatore conosce comunque il tipo e può verificarli, e inoltre confonde i programmatori.

Le variabili LOCALI dovrebbero avere nomi corti, e significativi. Se avete un qualsiasi contatore di ciclo, probabilmente sarà chiamato `i`. Chiamarlo `loop_counter` non è produttivo, non ci sono possibilità che `i` possa non essere capito. Analogamente, `tmp` può essere una qualsiasi variabile che viene usata per salvare temporaneamente un valore.

Se avete paura di fare casino coi nomi delle vostre variabili locali, allora avete un altro problema che è chiamato sindrome dello squilibrio dell'ormone della crescita delle funzioni. Vedere il capitolo 6 (funzioni).

## 5) Definizione di tipi (typedef)

Per favore non usate cose come `vps_t`. Usare il `typedef` per strutture e puntatori è uno **sbaglio**. Quando vedete:

```
vps_t a;
```

nei sorgenti, cosa significa? Se, invece, dicesse:

```
struct virtual_container *a;
```

potreste dire cos'è effettivamente `a`.

Molte persone pensano che la definizione dei tipi **migliori la leggibilità**. Non molto. Sono utili per:

- (a) gli oggetti completamente opachi (dove `typedef` viene proprio usato allo scopo di **nascondere** cosa sia davvero l'oggetto).

Esempio: `pte_t` eccetera sono oggetti opachi che potete usare solamente con le loro funzioni accessorie.

---

**Note:** Gli oggetti opachi e le funzioni accessorie non sono, di per se, una bella cosa. Il motivo per cui abbiamo cose come `pte_t` eccetera è che davvero non c'è alcuna informazione portabile.

- (b) i tipi chiaramente interi, dove l'astrazione **aiuta** ad evitare confusione sul fatto che siano `int` oppure `long`.

`u8/u16/u32` sono `typedef` perfettamente accettabili, anche se ricadono nella categoria (d) piuttosto che in questa.

---

**Note:**

Ancora - dev'esserci una **ragione** per farlo. Se qualcosa è `unsigned long`, non c'è alcun bisogno di avere:

```
typedef unsigned long myfalgs_t;
```

ma se ci sono chiare circostanze in cui potrebbe essere `unsigned int` e in altre configurazioni `unsigned long`, allora certamente `typedef` è una buona scelta.

- (c) quando di rado create letteralmente dei **nuovi** tipi su cui effettuare verifiche.
- (d) circostanze eccezionali, in cui si definiscono nuovi tipi identici a quelli definiti dallo standard C99.

Nonostante ci voglia poco tempo per abituare occhi e cervello all'uso dei tipi standard come `uint32_t`, alcune persone ne obiettano l'uso.

Perciò, i tipi specifici di Linux `u8/u16/u32/u64` e i loro equivalenti con segno, identici ai tipi standard, sono permessi- tuttavia, non sono obbligatori per il nuovo codice.

- (e) i tipi sicuri nella spazio utente.

In alcune strutture dati visibili dallo spazio utente non possiamo richiedere l'uso dei tipi C99 e nemmeno i vari `u32` descritti prima. Perciò, utilizziamo `_u32` e tipi simili in tutte le strutture dati condivise con lo spazio utente.

Magari ci sono altri casi validi, ma la regola di base dovrebbe essere di non usare MAI MAI un `typedef` a meno che non rientri in una delle regole descritte qui.

In generale, un puntatore, o una struttura a cui si ha accesso diretto in modo ragionevole, non dovrebbero **mai** essere definite con un `typedef`.

## 6) Funzioni

Le funzioni dovrebbero essere brevi e carine, e fare una cosa sola. Dovrebbero occupare uno o due schermi di testo (come tutti sappiamo, la dimensione di uno schermo secondo ISO/ANSI è di 80x24), e fare una cosa sola e bene.

La massima lunghezza di una funzione è inversamente proporzionale alla sua complessità e al livello di indentazione di quella funzione. Quindi, se avete una funzione che è concettualmente semplice ma che è implementata come un lunga (ma semplice) sequenza di caso-istruzione, dove avete molte piccole cose per molti casi differenti, allora va bene avere funzioni più lunghe.

Comunque, se avete una funzione complessa e sospettate che uno studente non particolarmente dotato del primo anno delle scuole superiori potrebbe non capire cosa faccia la funzione, allora dovreste attenervi strettamente ai limiti. Usate funzioni di supporto con nomi descrittivi (potete chiedere al compilatore di renderle inline se credete che sia necessario per le prestazioni, e probabilmente farà un lavoro migliore di quanto avreste potuto fare voi).

Un'altra misura delle funzioni sono il numero di variabili locali. Non dovrebbero eccedere le 5-10, oppure state sbagliando qualcosa. Ripensate la funzione, e dividetela in pezzettini. Generalmente, un cervello umano può seguire facilmente circa 7 cose diverse, di più lo confonderebbe. Lo sai d'essere brillante, ma magari vorresti riuscire a capire cos'avevi fatto due settimane prima.

Nei file sorgenti, separate le funzioni con una riga vuota. Se la funzione è esportata, la macro **EXPORT** per questa funzione deve seguire immediatamente la riga della parentesi graffa di chiusura. Ad esempio:

```
int system_is_up(void)
{
    return system_state == SYSTEM_RUNNING;
}
EXPORT_SYMBOL(system_is_up);
```

Nei prototipi di funzione, includete i nomi dei parametri e i loro tipi. Nonostante questo non sia richiesto dal linguaggio C, in Linux viene preferito perché è un modo semplice per aggiungere informazioni importanti per il lettore.

Non usate la parola chiave `extern` coi prototipi di funzione perché rende le righe più lunghe e non è strettamente necessario.

## 7) Centralizzare il ritorno delle funzioni

Sebbene sia deprecata da molte persone, l'istruzione goto è impiegata di frequente dai compilatori sotto forma di salto incondizionato.

L'istruzione goto diventa utile quando una funzione ha punti d'uscita multipli e vanno eseguite alcune procedure di pulizia in comune. Se non è necessario pulire alcunché, allora ritornate direttamente.

Assegnate un nome all'etichetta di modo che suggerisca cosa fa la goto o perché esiste. Un esempio di un buon nome potrebbe essere `out_free_buffer`: se la goto libera (free) un buffer. Evitate l'uso di nomi GW-BASIC come `err1`: ed `err2`:, potreste doverli riordinare se aggiungete o rimuovete punti d'uscita, e inoltre rende difficile verificarne la correttezza.

I motivo per usare le goto sono:

- i salti incondizionati sono più facili da capire e seguire
- l'annidamento si riduce
- si evita di dimenticare, per errore, di aggiornare un singolo punto d'uscita
- aiuta il compilatore ad ottimizzare il codice ridondante ;)

```
int fun(int a)
{
    int result = 0;
    char *buffer;

    buffer = kmalloc(SIZE, GFP_KERNEL);
    if (!buffer)
        return -ENOMEM;

    if (condition1) {
        while (loop1) {
            ...
        }
        result = 1;
        goto out_free_buffer;
    }
    ...

out_free_buffer:
    kfree(buffer);
    return result;
}
```

Un baco abbastanza comune di cui bisogna prendere nota è il one err bugs che assomiglia a questo:

```
err:
    kfree(foo->bar);
    kfree(foo);
    return ret;
```

Il baco in questo codice è che in alcuni punti d'uscita la variabile `foo` è `NULL`. Normalmente si corregge questo baco dividendo la gestione dell'errore in due parti `err_free_bar`: e `err_free_foo`:

```
err_free_bar:  
    kfree(foo->bar);  
err_free_foo:  
    kfree(foo);  
    return ret;
```

Idealmente, dovreste simulare condizioni d'errore per verificare i vostri percorsi d'uscita.

## 8) Commenti

I commenti sono una buona cosa, ma c'è anche il rischio di esagerare. MAI spiegare COME funziona il vostro codice in un commento: è molto meglio scrivere il codice di modo che il suo funzionamento sia ovvio, inoltre spiegare codice scritto male è una perdita di tempo.

Solitamente, i commenti devono dire COSA fa il codice, e non COME lo fa. Inoltre, cercate di evitare i commenti nel corpo della funzione: se la funzione è così complessa che dovete commentarla a pezzi, allora dovreste tornare al punto 6 per un momento. Potete mettere dei piccoli commenti per annotare o avvisare il lettore circa un qualcosa di particolarmente arguto (o brutto), ma cercate di non esagerare. Invece, mettete i commenti in testa alla funzione spiegando alle persone cosa fa, e possibilmente anche il PERCHÉ.

Per favore, quando commentate una funzione dell'API del kernel usate il formato kernel-doc. Per maggiori dettagli, leggete i file in :ref:[Documentation/translations/it\\_IT/doc-guide/](#) e in [script/kernel-doc](#).

Lo stile preferito per i commenti più lunghi (multi-riga) è:

```
/*  
 * This is the preferred style for multi-line  
 * comments in the Linux kernel source code.  
 * Please use it consistently.  
 *  
 * Description: A column of asterisks on the left side,  
 * with beginning and ending almost-blank lines.  
 */
```

Per i file in net/ e in drivers/net/ lo stile preferito per i commenti più lunghi (multi-riga) è leggermente diverso.

```
/* The preferred comment style for files in net/ and drivers/net  
 * looks like this.  
 *  
 * It is nearly the same as the generally preferred comment style,  
 * but there is no initial almost-blank line.  

```

È anche importante commentare i dati, sia per i tipi base che per tipi derivati. A questo scopo, dichiarate un dato per riga (niente virgole per una dichiarazione multipla). Questo vi lascerà spazio per un piccolo commento per spiegarne l'uso.

## 9) Avete fatto un pasticcio

Va bene, li facciamo tutti. Probabilmente vi è stato detto dal vostro aiutante Unix di fiducia che GNU emacs formatta automaticamente il codice C per conto vostro, e avete notato che sì, in effetti lo fa, ma che i modi predefiniti non sono proprio allettanti (infatti, sono peggio che premere tasti a caso - un numero infinito di scimmie che scrivono in GNU emacs non faranno mai un buon programma).

Quindi, potete sbarazzarvi di GNU emacs, o riconfigurarlo con valori più sensati. Per fare quest'ultima cosa, potete appiccicare il codice che segue nel vostro file .emacs:

```
(defun c-lineup-arglist-tabs-only (ignored)
  "Line up argument lists by tabs, not spaces"
  (let* ((anchor (c-langelem-pos c-syntactic-element))
         (column (c-langelem-2nd-pos c-syntactic-element))
         (offset (- (1+ column) anchor)))
    (steps (floor offset c-basic-offset)))
  (* (max steps 1)
     c-basic-offset))

(dir-locals-set-class-variables
 'linux-kernel
 '((c-mode .
  (c-basic-offset . 8)
  (c-label-minimum-indentation . 0)
  (c-offsets-alist .
   (arglist-close . c-lineup-arglist-tabs-only)
   (arglist-cont-nonempty .
    (c-lineup-gcc-asm-reg c-lineup-arglist-tabs-only))
   (arglist-intro . +)
   (brace-list-intro . +)
   (c . c-lineup-C-comments)
   (case-label . 0)
   (comment-intro . c-lineup-comment)
   (cpp-define-intro . +)
   (cpp-macro . -1000)
   (cpp-macro-cont . +)
   (defun-block-intro . +)
   (else-clause . 0)
   (func-decl-cont . +)
   (inclass . +)
   (inher-cont . c-lineup-multi-inher)
   (knr-argdecl-intro . 0)
   (label . -1000)
   (statement . 0)
   (statement-block-intro . +)
   (statement-case-intro . +)
   (statement-cont . +)
   (substatement . +)
   )))
  (indent-tabs-mode . t))
```

```
(show-trailing-whitespace . t)
)))

(dir-locals-set-directory-class
 (expand-file-name "~/src/linux-trees")
 'linux-kernel)
```

Questo farà funzionare meglio emacs con lo stile del kernel per i file che si trovano nella cartella `~/src/linux-trees`.

Ma anche se dovete fallire nell'ottenere una formattazione sensata in emacs non tutto è perduto: usate `indent`.

Ora, ancora, GNU `indent` ha la stessa configurazione decerebrata di GNU emacs, ed è per questo che dovete passargli alcune opzioni da riga di comando. Tuttavia, non è così terribile, perché perfino i creatori di GNU `indent` riconoscono l'autorità di K&R (le persone del progetto GNU non sono cattive, sono solo mal indirizzate sull'argomento), quindi date ad `indent` le opzioni `-kr -i8` (che significa K&R, 8 caratteri di indentazione), o utilizzate `scripts/Lindent` che indenterà usando l'ultimo stile.

`indent` ha un sacco di opzioni, e specialmente quando si tratta di riformattare i commenti dovreste dare un'occhiata alle pagine man. Ma ricordatevi: `indent` non è un correttore per una cattiva programmazione.

Da notare che potete utilizzare anche `clang-format` per aiutarvi con queste regole, per riformattare rapidamente ad automaticamente alcune parti del vostro codice, e per revisionare interi file al fine di identificare errori di stile, refusi e possibilmente anche delle migliorie. È anche utile per ordinare gli `#include`, per allineare variabili/macro, per ridistribuire il testo e altre cose simili. Per maggiori dettagli, consultate il file [Documentation/translations/it\\_IT/process/clang-format.rst](#).

## 10) File di configurazione Kconfig

Per tutti i file di configurazione Kconfig\* che si possono trovare nei sorgenti, l'indentazione è un po' differente. Le linee dopo un `config` sono indentate con un tab, mentre il testo descrittivo è indentato di ulteriori due spazi. Esempio:

```
config AUDIT
    bool "Auditing support"
    depends on NET
    help
        Enable auditing infrastructure that can be used with another
        kernel subsystem, such as SELinux (which requires this for
        logging of avc messages output). Does not do system-call
        auditing without CONFIG_AUDITSYSCALL.
```

Le funzionalità davvero pericolose (per esempio il supporto alla scrittura per certi filesystem) dovrebbero essere dichiarate chiaramente come tali nella stringa di titolo:

```
config ADFS_FS_RW
    bool "ADFS write support (DANGEROUS)"
```

```
depends on ADFS_FS
```

```
...
```

Per la documentazione completa sui file di configurazione, consultate il documento Documentation/kbuild/kconfig-language.rst

## 11) Strutture dati

Le strutture dati che hanno una visibilità superiore al contesto del singolo thread in cui vengono create e distrutte, dovrebbero sempre avere un contatore di riferimenti. Nel kernel non esiste un *garbage collector* (e fuori dal kernel i *garbage collector* sono lenti e inefficienti), questo significa che **dovete** assolutamente avere un contatore di riferimenti per ogni cosa che usate.

Avere un contatore di riferimenti significa che potete evitare la sincronizzazione e permette a più utenti di accedere alla struttura dati in parallelo - e non doversi preoccupare di una struttura dati che improvvisamente sparisce dalla loro vista perché il loro processo dormiva o stava facendo altro per un attimo.

Da notare che la sincronizzazione **non** si sostituisce al conteggio dei riferimenti. La sincronizzazione ha lo scopo di mantenere le strutture dati coerenti, mentre il conteggio dei riferimenti è una tecnica di gestione della memoria. Solitamente servono entrambe le cose, e non vanno confuse fra di loro.

Quando si hanno diverse classi di utenti, le strutture dati possono avere due livelli di contatori di riferimenti. Il contatore di classe conta il numero dei suoi utenti, e il contatore globale viene decrementato una sola volta quando il contatore di classe va a zero.

Un esempio di questo tipo di conteggio dei riferimenti multi-livello può essere trovato nella gestore della memoria (`struct mm_struct: mm_user e mm_count`), e nel codice dei filesystem (`struct super_block: s_count e s_active`).

Ricordatevi: se un altro thread può trovare la vostra struttura dati, e non avete un contatore di riferimenti per essa, quasi certamente avete un baco.

## 12) Macro, enumerati e RTL

I nomi delle macro che definiscono delle costanti e le etichette degli enumerati sono scritte in maiuscolo.

```
#define CONSTANT 0x12345
```

Gli enumerati sono da preferire quando si definiscono molte costanti correlate.

I nomi delle macro in MAIUSCOLO sono preferibili ma le macro che assomigliano a delle funzioni possono essere scritte in minuscolo.

Generalmente, le funzioni inline sono preferibili rispetto alle macro che sembrano funzioni.

Le macro che contengono più istruzioni dovrebbero essere sempre chiuse in un blocco do - while:

```
#define macrofun(a, b, c)
    do {
```

```
|
```

```
if (a == 5) |  
    do_this(b, c); |  
} while (0)
```

Cose da evitare quando si usano le macro:

- 1) le macro che hanno effetti sul flusso del codice:

```
#define FOO(x) |  
    do { |  
        if (blah(x) < 0) |  
            return -EBUGGERED; |  
    } while (0)
```

sono **proprio** una pessima idea. Sembra una chiamata a funzione ma termina la funzione chiamante; non cercate di rompere il decodificatore interno di chi legge il codice.

- 2) le macro che dipendono dall'uso di una variabile locale con un nome magico:

```
#define FOO(val) bar(index, val)
```

potrebbe sembrare una bella cosa, ma è dannatamente confusionario quando uno legge il codice e potrebbe romperlo con una cambiamento che sembra innocente.

3) le macro con argomenti che sono utilizzati come l-values; questo potrebbe ritorcervisi contro se qualcuno, per esempio, trasforma FOO in una funzione inline.

4) dimenticatevi delle precedenze: le macro che definiscono espressioni devono essere racchiuse fra parentesi. State attenti a problemi simili con le macro parametrizzate.

```
#define CONSTANT 0x4000  
#define CONSTEXP (CONSTANT | 3)
```

5) collisione nello spazio dei nomi quando si definisce una variabile locale in una macro che sembra una funzione:

```
#define FOO(x) |  
{ |  
    typeof(x) ret; |  
    ret = calc_ret(x); |  
    (ret); |  
}
```

ret è un nome comune per una variabile locale - \_foo\_ret difficilmente andrà in conflitto con una variabile già esistente.

Il manuale di cpp si occupa esaustivamente delle macro. Il manuale di sviluppo di gcc copre anche l'RTL che viene usato frequentemente nel kernel per il linguaggio assembler.

## 13) Visualizzare i messaggi del kernel

Agli sviluppatori del kernel piace essere visti come dotti. Tenete un occhio di riguardo per l'ortografia e farete una belle figura. In inglese, evitate l'uso incorretto di abbreviazioni come `dont:` usate `do not` oppure `don't`. Scrivete messaggi concisi, chiari, e inequivocabili.

I messaggi del kernel non devono terminare con un punto fermo.

Scrivere i numeri fra parentesi (%d) non migliora alcunché e per questo dovrebbero essere evitati.

Ci sono alcune macro per la diagnostica in `<linux/device.h>` che dovreste usare per assicurarvi che i messaggi vengano associati correttamente ai dispositivi e ai driver, e che siano etichettati correttamente: `dev_err()`, `dev_warn()`, `dev_info()`, e così via. Per messaggi che non sono associati ad alcun dispositivo, `<linux/printk.h>` definisce `pr_info()`, `pr_warn()`, `pr_err()`, eccetera.

Tirar fuori un buon messaggio di debug può essere una vera sfida; e quando l'avete può essere d'enorme aiuto per risolvere problemi da remoto. Tuttavia, i messaggi di debug sono gestiti differentemente rispetto agli altri. Le funzioni `pr_XXX()` stampano incondizionatamente ma `pr_debug()` no; essa non viene compilata nella configurazione predefinita, a meno che `DEBUG` o `CONFIG_DYNAMIC_DEBUG` non vengono impostati. Questo vale anche per `dev_dbg()` e in aggiunta `VERBOSE_DEBUG` per aggiungere i messaggi `dev_vdbg()`.

Molti sottosistemi hanno delle opzioni di debug in Kconfig che aggiungono `-DDEBUG` nei corrispettivi Makefile, e in altri casi aggiungono `#define DEBUG` in specifici file. Infine, quando un messaggio di debug dev'essere stampato incondizionatamente, per esempio perché siete già in una sezione di debug racchiusa in `#ifdef`, potete usare `printk(KERN_DEBUG ...)`.

## 14) Assegnare memoria

Il kernel fornisce i seguenti assegnatori ad uso generico: `kmalloc()`, `kzalloc()`, `kmalloc_array()`, `kcalloc()`, `vmalloc()`, e `vzalloc()`. Per maggiori informazioni, consultate la documentazione dell'API: Documentation/translations/it\_IT/core-api/memory-allocation.rst

Il modo preferito per passare la dimensione di una struttura è il seguente:

```
p = kmalloc(sizeof(*p), ...);
```

La forma alternativa, dove il nome della struttura viene scritto interamente, peggiora la leggibilità e introduce possibili bachi quando il tipo di puntatore cambia tipo ma il corrispondente `sizeof` non viene aggiornato.

Il valore di ritorno è un puntatore void, effettuare un cast su di esso è ridondante. La conversione fra un puntatore void e un qualsiasi altro tipo di puntatore è garantito dal linguaggio di programmazione C.

Il modo preferito per assegnare un vettore è il seguente:

```
p = kmalloc_array(n, sizeof(...), ...);
```

Il modo preferito per assegnare un vettore a zero è il seguente:

```
p = kcalloc(n, sizeof(...), ...);
```

Entrambe verificano la condizione di overflow per la dimensione d'assegnamento `n * sizeof(...)`, se accade ritorneranno `NULL`.

Questi allocatori generici producono uno *stack dump* in caso di fallimento a meno che non venga esplicitamente specificato `_GFP_NOWARN`. Quindi, nella maggior parte dei casi, è inutile stampare messaggi aggiuntivi quando uno di questi allocatori ritornano un puntatore `NULL`.

### 15) Il morbo inline

Sembra che ci sia la percezione errata che gcc abbia una qualche magica opzione “rendimi più veloce” chiamata `inline`. In alcuni casi l’uso di `inline` è appropriato (per esempio in sostituzione delle macro, vedi capitolo 12), ma molto spesso non lo è. L’uso abbondante della parola chiave `inline` porta ad avere un kernel più grande, che si traduce in un sistema nel suo complesso più lento per via di una cache per le istruzioni della CPU più grande e poi semplicemente perché ci sarà meno spazio disponibile per una pagina di cache. Pensateci un attimo; una fallimento nella cache causa una ricerca su disco che può tranquillamente richiedere 5 millisecondi. Ci sono TANTI cicli di CPU che potrebbero essere usati in questi 5 millisecondi.

Spesso le persone dicono che aggiungere `inline` a delle funzioni dichiarate `static` e utilizzare una sola volta è sempre una scelta vincente perché non ci sono altri compromessi. Questo è tecnicamente vero ma gcc è in grado di trasformare automaticamente queste funzioni in `inline`; i problemi di manutenzione del codice per rimuovere gli `inline` quando compare un secondo utente surclassano il potenziale vantaggio nel suggerire a gcc di fare una cosa che avrebbe fatto comunque.

### 16) Nomi e valori di ritorno delle funzioni

Le funzioni possono ritornare diversi tipi di valori, e uno dei più comuni è quel valore che indica se una funzione ha completato con successo o meno. Questo valore può essere rappresentato come un codice di errore intero (-E`xxx` = fallimento, 0 = successo) oppure un booleano di successo (0 = fallimento, non-zero = successo).

Mischiare questi due tipi di rappresentazioni è un terreno fertile per i bachi più insidiosi. Se il linguaggio C includesse una forte distinzione fra gli interi e i booleani, allora il compilatore potrebbe trovare questi errori per conto nostro ... ma questo non c’è. Per evitare di imbattersi in questo tipo di baco, seguite sempre la seguente convenzione:

Se il nome di una funzione è un’azione o un comando imperativo, essa dovrebbe ritornare un codice di errore intero. Se il nome è un predicato, la funzione dovrebbe ritornare un booleano di “successo”

Per esempio, `add_work` è un comando, e la funzione `add_work()` ritorna 0 in caso di successo o -EBUSY in caso di fallimento. Allo stesso modo, `PCI device present` è un predicato, e la funzione `pci_dev_present()` ritorna 1 se trova il dispositivo corrispondente con successo, altrimenti 0.

Tutte le funzioni esportate (EXPORT) devono rispettare questa convenzione, e così dovrebbero anche tutte le funzioni pubbliche. Le funzioni private (`static`) possono non seguire questa convenzione, ma è comunque raccomandato che lo facciano.

Le funzioni il cui valore di ritorno è il risultato di una computazione, piuttosto che l'indicazione sul successo di tale computazione, non sono soggette a questa regola. Solitamente si indicano gli errori ritornando un qualche valore fuori dai limiti. Un tipico esempio è quello delle funzioni che ritornano un puntatore; queste utilizzano NULL o ERR\_PTR come meccanismo di notifica degli errori.

## 17) L'uso di bool

Nel kernel Linux il tipo bool deriva dal tipo \_Bool dello standard C99. Un valore bool può assumere solo i valori 0 o 1, e implicitamente o esplicitamente la conversione a bool converte i valori in vero (*true*) o falso (*false*). Quando si usa un tipo bool il costrutto `!!` non sarà più necessario, e questo va ad eliminare una certa serie di bachi.

Quando si usano i valori booleani, dovreste utilizzare le definizioni di *true* e *false* al posto dei valori 1 e 0.

Per il valore di ritorno delle funzioni e per le variabili sullo stack, l'uso del tipo bool è sempre appropriato. L'uso di bool viene incoraggiato per migliorare la leggibilità e spesso è molto meglio di 'int' nella gestione di valori booleani.

Non usate bool se per voi sono importanti l'ordine delle righe di cache o la loro dimensione; la dimensione e l'allineamento cambia a seconda dell'architettura per la quale è stato compilato. Le strutture che sono state ottimizzate per l'allineamento o la dimensione non dovrebbero usare bool.

Se una struttura ha molti valori true/false, considerate l'idea di raggrupparli in un intero usando campi da 1 bit, oppure usate un tipo dalla larghezza fissa, come u8.

Come per gli argomenti delle funzioni, molti valori true/false possono essere raggruppati in un singolo argomento a bit denominato 'flags'; spesso 'flags' è un'alternativa molto più leggibile se si hanno valori costanti per true/false.

Detto ciò, un uso parsimonioso di bool nelle strutture dati e negli argomenti può migliorare la leggibilità.

## 18) Non reinventate le macro del kernel

Il file di intestazione include/linux/kernel.h contiene un certo numero di macro che dovreste usare piuttosto che implementarne una qualche variante. Per esempio, se dovete calcolare la lunghezza di un vettore, sfruttate la macro:

```
#define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))
```

Analogamente, se dovete calcolare la dimensione di un qualche campo di una struttura, usate

```
#define sizeof_field(t, f) (sizeof(((t*)0)->f))
```

Ci sono anche le macro min() e max() che, se vi serve, effettuano un controllo rigido sui tipi. Sentitevi liberi di leggere attentamente questo file d'intestazione per scoprire cos'altro è stato definito che non dovreste reinventare nel vostro codice.

### 19) Linee di configurazione degli editor e altre schifezze

Alcuni editor possono interpretare dei parametri di configurazione integrati nei file sorgenti e indicati con dai marcatori speciali. Per esempio, emacs interpreta le linee marcate nel seguente modo:

```
- *- mode: c -*-
```

O come queste:

```
/*
Local Variables:
compile-command: "gcc -DMAGIC_DEBUG_FLAG foo.c"
End:
*/
```

Vim interpreta i marcatori come questi:

```
/* vim:set sw=8 noet */
```

Non includete nessuna di queste cose nei file sorgenti. Le persone hanno le proprie configurazioni personali per l'editor, e i vostri sorgenti non dovrebbero sovrascrivergliele. Questo vale anche per i marcatori d'indentazione e di modalità d'uso. Le persone potrebbero aver configurato una modalità su misura, oppure potrebbero avere qualche altra magia per far funzionare bene l'indentazione.

### 20) Inline assembly

Nel codice specifico per un'architettura, potreste aver bisogno di codice *inline assembly* per interfacciarsi col processore o con una funzionalità specifica della piattaforma. Non esitate a farlo quando è necessario. Comunque, non usatele gratuitamente quando il C può fare la stessa cosa. Potete e dovreste punzecchiare l'hardware in C quando è possibile.

Considerate la scrittura di una semplice funzione che racchiude pezzi comuni di codice assembler piuttosto che continuare a riscrivere delle piccole varianti. Ricordatevi che l'*inline assembly* può utilizzare i parametri C.

Il codice assembler più corposo e non banale dovrebbe andare nei file .S, coi rispettivi prototipi C definiti nei file d'intestazione. I prototipi C per le funzioni assembler dovrebbero usare `asm linkage`.

Potreste aver bisogno di marcare il vostro codice asm come volatile al fine d'evitare che GCC lo rimuova quando pensa che non ci siano effetti collaterali. Non c'è sempre bisogno di farlo, e farlo quando non serve limita le ottimizzazioni.

Quando scrivete una singola espressione *inline assembly* contenente più istruzioni, mettete ognuna di queste istruzioni in una stringa e riga diversa; ad eccezione dell'ultima stringa/istruzione, ognuna deve terminare con `\n\t` al fine di allineare correttamente l'assembler che verrà generato:

```
asm ("magic %reg1, #42\n\t"
      "more_magic %reg2, %reg3"
      : /* outputs */ : /* inputs */ : /* clobbers */);
```

## 21) Compilazione sotto condizione

Ovunque sia possibile, non usate le direttive condizionali del preprocessore (#if, #ifdef) nei file .c; farlo rende il codice difficile da leggere e da seguire. Invece, usate queste direttive nei file d'intestazione per definire le funzioni usate nei file .c, fornendo i relativi stub nel caso #else, e quindi chiamate queste funzioni senza condizioni di preprocessore. Il compilatore non produrrà alcun codice per le funzioni stub, produrrà gli stessi risultati, e la logica rimarrà semplice da seguire.

È preferibile non compilare intere funzioni piuttosto che porzioni d'esse o porzioni d'espressioni. Piuttosto che mettere una ifdef in un'espressione, fattorizzate parte dell'espressione, o interamente, in funzioni e applicate la direttiva condizionale su di esse.

Se avete una variabile o funzione che potrebbe non essere usata in alcune configurazioni, e quindi il compilatore potrebbe avvisarvi circa la definizione inutilizzata, marcate questa definizione come `_maybe_unused` piuttosto che racchiuderla in una direttiva condizionale del preprocessore. (Comunque, se una variabile o funzione è *sempre* inutilizzata, rimuovetela).

Nel codice, dov'è possibile, usate la macro `IS_ENABLED` per convertire i simboli Kconfig in espressioni booleane C, e quindi usatela nelle classiche condizioni C:

```
if (IS_ENABLED(CONFIG_SOMETHING)) {
    ...
}
```

Il compilatore valuterà la condizione come costante (constant-fold), e quindi includerà o escluderà il blocco di codice come se fosse in un `#ifdef`, quindi non ne aumenterà il tempo di esecuzione. Tuttavia, questo permette al compilatore C di vedere il codice nel blocco condizionale e verificarne la correttezza (sintassi, tipi, riferimenti ai simboli, eccetera). Quindi dovete comunque utilizzare `#ifdef` se il codice nel blocco condizionale esiste solo quando la condizione è soddisfatta.

Alla fine di un blocco corposo di `#if` o `#ifdef` (più di alcune linee), mettete un commento sulla stessa riga di `#endif`, annotando la condizione che termina. Per esempio:

```
#ifdef CONFIG_SOMETHING
...
#endif /* CONFIG_SOMETHING */
```

## Appendice I) riferimenti

The C Programming Language, Second Edition by Brian W. Kernighan and Dennis M. Ritchie. Prentice Hall, Inc., 1988. ISBN 0-13-110362-8 (paperback), 0-13-110370-9 (hardback).

The Practice of Programming by Brian W. Kernighan and Rob Pike. Addison-Wesley, Inc., 1999. ISBN 0-201-61586-X.

Manuali GNU - nei casi in cui sono compatibili con K&R e questo documento - per indent, cpp, gcc e i suoi dettagli interni, tutto disponibile qui <http://www.gnu.org/manual/>

WG14 è il gruppo internazionale di standardizzazione per il linguaggio C, URL: <http://www.open-std.org/JTC1/SC22/WG14/>

Kernel process/coding-style.rst, by [greg@kroah.com](mailto:greg@kroah.com) at OLS 2002: [http://www.kroah.com/linux/talks/ols\\_2002\\_kernel\\_codingstyle\\_talk/html/](http://www.kroah.com/linux/talks/ols_2002_kernel_codingstyle_talk/html/)

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/maintainer-pgp-guide.rst

**Translator** Alessia Mantegazza <[amanantegazza@vaga.pv.it](mailto:amanantegazza@vaga.pv.it)>

### La guida a PGP per manutentori del kernel

**Author** Konstantin Ryabitsev <[konstantin@linuxfoundation.org](mailto:konstantin@linuxfoundation.org)>

Questo documento è destinato agli sviluppatori del kernel Linux, in particolar modo ai manutentori. Contiene degli approfondimenti riguardo informazioni che sono state affrontate in maniera più generale nella sezione “[Protecting Code Integrity](#)” pubblicata dalla Linux Foundation. Per approfondire alcuni argomenti trattati in questo documento è consigliato leggere il documento sopraindicato

### Il ruolo di PGP nello sviluppo del kernel Linux

PGP aiuta ad assicurare l'integrità del codice prodotto dalla comunità di sviluppo del kernel e, in secondo luogo, stabilisce canali di comunicazione affidabili tra sviluppatori attraverso lo scambio di email firmate con PGP.

Il codice sorgente del kernel Linux è disponibile principalmente in due formati:

- repository distribuiti di sorgenti (git)
- rilasci periodici di istantanee (archivi tar)

Sia i repository git che gli archivi tar portano le firme PGP degli sviluppatori che hanno creato i rilasci ufficiali del kernel. Queste firme offrono una garanzia crittografica che le versioni scaricabili rese disponibili via [kernel.org](http://kernel.org), o altri portali, siano identiche a quelle che gli sviluppatori hanno sul loro posto di lavoro. A tal scopo:

- i repository git forniscono firme PGP per ogni tag
- gli archivi tar hanno firme separate per ogni archivio

### Fidatevi degli sviluppatori e non dell'infrastruttura

Fin dal 2011, quando i sistemi di [kernel.org](http://kernel.org) furono compromessi, il principio generale del progetto Kernel Archives è stato quello di assumere che qualsiasi parte dell'infrastruttura possa essere compromessa in ogni momento. Per questa ragione, gli amministratori hanno intrapreso deliberatamente dei passi per enfatizzare che la fiducia debba risiedere sempre negli sviluppatori e mai nel codice che gestisce l'infrastruttura, indipendentemente da quali che siano le pratiche di sicurezza messe in atto.

Il principio sopra indicato è la ragione per la quale è necessaria questa guida. Vogliamo essere sicuri che il riporre la fiducia negli sviluppatori non sia fatto semplicemente per incolpare qualcun'altro per future falle di sicurezza. L'obiettivo è quello di fornire una serie di linee guida che gli sviluppatori possano seguire per creare un ambiente di lavoro sicuro e salvaguardare le chiavi PGP usate nello stabilire l'integrità del kernel Linux stesso.

## Strumenti PGP

### Usare GnuPG v2

La vostra distribuzione potrebbe avere già installato GnuPG, dovete solo verificare che stia utilizzando la versione 2.x e non la serie 1.4 - molte distribuzioni forniscono entrambe, di base il comando "gpg" invoca GnuPG v.1. Per controllate usate:

```
$ gpg --version | head -n1
```

Se visualizzate gpg (GnuPG) 1.4.x, allora state usando GnuPG v.1. Provate il comando gpg2 (se non lo avete, potreste aver bisogno di installare il pacchetto gnupg2):

```
$ gpg2 --version | head -n1
```

Se visualizzate gpg (GnuPG) 2.x.x, allora siete pronti a partire. Questa guida assume che abbiate la versione 2.2.(o successiva) di GnuPG. Se state usando la versione 2.0, alcuni dei comandi indicati qui non funzioneranno, in questo caso considerate un aggiornamento all'ultima versione, la 2.2. Versioni di gnupg-2.1.11 e successive dovrebbero essere compatibili per gli obiettivi di questa guida.

Se avete entrambi i comandi: gpg e gpg2, assicuratevi di utilizzare sempre la versione V2, e non quella vecchia. Per evitare errori potreste creare un alias:

```
$ alias gpg=gpg2
```

Potete mettere questa opzione nel vostro .bashrc in modo da essere sicuri.

### Configurare le opzioni di gpg-agent

L'agente GnuPG è uno strumento di aiuto che partirà automaticamente ogni volta che userete il comando gpg e funzionerà in background con l'obiettivo di individuare la passphrase. Ci sono due opzioni che dovreste conoscere per personalizzare la scadenza della passphrase nella cache:

- **default-cache-ttl** (secondi): Se usate ancora la stessa chiave prima che il time-to-live termini, il conto alla rovescia si resetterà per un altro periodo. Di base è di 600 (10 minuti).
- **max-cache-ttl** (secondi): indipendentemente da quanto sia recente l'ultimo uso della chiave da quando avete inserito la passphrase, se il massimo time-to-live è scaduto, dovrete reinserire nuovamente la passphrase. Di base è di 30 minuti.

Se ritenete entrambe questi valori di base troppo corti (o troppo lunghi), potete creare il vostro file `~/.gnupg/gpg-agent.conf` ed impostare i vostri valori:

```
# set to 30 minutes for regular ttl, and 2 hours for max ttl
default-cache-ttl 1800
max-cache-ttl 7200
```

---

**Note:** Non è più necessario far partire l'agente gpg manualmente all'inizio della vostra sessione. Dovreste controllare i file rc per rimuovere tutto ciò che riguarda vecchie le versioni di GnuPG, poiché potrebbero non svolgere più bene il loro compito.

---

### Impostare un *refresh* con cronjob

Potreste aver bisogno di rinfrescare regolarmente il vostro portachiavi in modo aggiornare le chiavi pubbliche di altre persone, lavoro che è svolto al meglio con un cronjob giornaliero:

```
@daily /usr/bin/gpg2 --refresh >/dev/null 2>&1
```

Controllate il percorso assoluto del vostro comando gpg o gpg2 e usate il comando gpg2 se per voi gpg corrisponde alla versione GnuPG v.1.

### Proteggere la vostra chiave PGP primaria

Questa guida parte dal presupposto che abbiate già una chiave PGP che usate per lo sviluppo del kernel Linux. Se non ne avete ancora una, date uno sguardo al documento “[Protecting Code Integrity](#)” che abbiamo menzionato prima.

Dovreste inoltre creare una nuova chiave se quella attuale è inferiore a 2048 bit (RSA).

### Chiave principale o sottochiavi

Le sottochiavi sono chiavi PGP totalmente indipendenti, e sono collegate alla chiave principale attraverso firme certificate. È quindi importante comprendere i seguenti punti:

1. Non ci sono differenze tecniche tra la chiave principale e la sottochiave.
2. In fesa di creazione, assegniamo limitazioni funzionali ad ogni chiave assegnando capacità specifiche.
3. Una chiave PGP può avere 4 capacità:
  - **[S]** può essere usata per firmare
  - **[E]** può essere usata per criptare
  - **[A]** può essere usata per autenticare
  - **[C]** può essere usata per certificare altre chiavi
4. Una singola chiave può avere più capacità
5. Una sottochiave è completamente indipendente dalla chiave principale. Un messaggio criptato con la sottochiave non può essere decriptato con quella principale. Se perdete la vostra sottochiave privata, non può essere rigenerata in nessun modo da quella principale.

La chiave con capacità **[C]** (certify) è identificata come la chiave principale perché è l'unica che può essere usata per indicare la relazione con altre chiavi. Solo la chiave **[C]** può essere usata per:

- Aggiungere o revocare altre chiavi (sottochiavi) che hanno capacità S/E/A
- Aggiungere, modificare o eliminare le identità (uids) associate alla chiave
- Aggiungere o modificare la data di termine di sé stessa o di ogni sottochiave
- Firmare le chiavi di altre persone a scopo di creare una rete di fiducia

Di base, alla creazione di nuove chiavi, GnuPG genera quanto segue:

- Una chiave madre che porta sia la capacità di certificazione che quella di firma (**[SC]**)
- Una sottochiave separata con capacità di criptaggio (**[E]**)

Se avete usato i parametri di base per generare la vostra chiave, quello sarà il risultato. Potete verificarlo utilizzando `gpg --list-secret-keys`, per esempio:

```
sec    rsa2048 2018-01-23 [SC] [expires: 2020-01-23]
      00000000000000000000000000000000AAAABBBBCCCCDDDD
uid          [ultimate] Alice Dev <audev@kernel.org>
ssb    rsa2048 2018-01-23 [E] [expires: 2020-01-23]
```

Qualsiasi chiave che abbia la capacità **[C]** è la vostra chiave madre, indipendentemente da quali altre capacità potreste averle assegnato.

La lunga riga sotto la voce `sec` è la vostra impronta digitale – negli esempi che seguono, quando vedere `[fpr]` ci si riferisce a questa stringa di 40 caratteri.

### Assicuratevi che la vostra passphrase sia forte

GnuPG utilizza le passphrases per criptare la vostra chiave privata prima di salvarla sul disco. In questo modo, anche se il contenuto della vostra cartella `.gnupg` venisse letto o trafugato nella sua interezza, gli attaccanti non potrebbero comunque utilizzare le vostre chiavi private senza aver prima ottenuto la passphrase per decriptarle.

È assolutamente essenziale che le vostre chiavi private siano protette da una passphrase forte. Per impostarla o cambiarla, usate:

```
$ gpg --change-passphrase [fpr]
```

### Create una sottochiave di firma separata

Il nostro obiettivo è di proteggere la chiave primaria spostandola su un dispositivo sconnesso dalla rete, dunque se avete solo una chiave combinata **[SC]** allora dovreste creare una sottochiave di firma separata:

```
$ gpg --quick-add-key [fpr] ed25519 sign
```

Ricordate di informare il keyserver del vostro cambiamento, cosicché altri possano ricevere la vostra nuova sottochiave:

```
$ gpg --send-key [fpr]
```

---

**Note:** Supporto ECC in GnuPG GnuPG 2.1 e successivi supportano pienamente *Elliptic Curve Cryptography*, con la possibilità di combinare sottochiavi ECC con le tradizionali chiavi primarie RSA. Il principale vantaggio della crittografia ECC è che è molto più veloce da calcolare e crea firme più piccole se confrontate byte per byte con le chiavi RSA a più di 2048 bit. A meno che non pensiate di utilizzare un dispositivo smartcard che non supporta le operazioni ECC, vi raccomandiamo di creare sottochiavi di firma ECC per il vostro lavoro col kernel.

Se per qualche ragione preferite rimanere con sottochiavi RSA, nel comando precedente, sostituite “ed25519” con “rsa2048”. In aggiunta, se avete intenzione di usare un dispositivo hardware che non supporta le chiavi ED25519 ECC, come la Nitrokey Pro o la Yubikey, allora dovreste usare “nistp256” al posto di “ed25519”.

---

### Copia di riserva della chiave primaria per gestire il recupero da disastro

Maggiori sono le firme di altri sviluppatori che vengono applicate alla vostra, maggiori saranno i motivi per avere una copia di riserva che non sia digitale, al fine di effettuare un recupero da disastro.

Il modo migliore per creare una copia fisica della vostra chiave privata è l’uso del programma paperkey. Consultate `man paperkey` per maggiori dettagli sul formato dell’output ed i suoi punti di forza rispetto ad altre soluzioni. Paperkey dovrebbe essere già pacchettizzato per la maggior parte delle distribuzioni.

Eseguite il seguente comando per creare una copia fisica di riserva della vostra chiave privata:

```
$ gpg --export-secret-key [fpr] | paperkey -o /tmp/key-backup.txt
```

Stampate il file (o fate un pipe direttamente verso lpr), poi prendete una penna e scrivete la passphrase sul margine del foglio. **Questo è caldamente consigliato** perché la copia cartacea è comunque criptata con la passphrase, e se mai doveste cambiarla non vi ricorderete qual’era al momento della creazione di quella copia - *garantito*.

Mettete la copia cartacea e la passphrase scritta a mano in una busta e mettetela in un posto sicuro e ben protetto, preferibilmente fuori casa, magari in una cassetta di sicurezza in banca.

---

**Note:** Probabilmente la vostra stampante non è più quello stupido dispositivo connesso alla porta parallela, ma dato che il suo output è comunque criptato con la passphrase, eseguire la stampa in un sistema “cloud” moderno dovrebbe essere comunque relativamente sicuro. Un’opzione potrebbe essere quella di cambiare la passphrase della vostra chiave primaria subito dopo aver finito con paperkey.

---

## Copia di riserva di tutta la cartella GnuPG

**Warning: !!!Non saltate questo passo!!!**

Quando avete bisogno di recuperare le vostre chiavi PGP è importante avere una copia di riserva pronta all'uso. Questo sta su un diverso piano di prontezza rispetto al recupero da disastro che abbiamo risolto con paperkey. Vi affiderete a queste copie esterne quando dovrete usare la vostra chiave Certify – ovvero quando fate modifiche alle vostre chiavi o firmate le chiavi di altre persone ad una conferenza o ad un gruppo d'incontro.

Incominciate con una piccola chiavetta di memoria USB (preferibilmente due) che userete per le copie di riserva. Dovrete criptarle usando LUKS – fate riferimento alla documentazione della vostra distribuzione per capire come fare.

Per la passphrase di criptazione, potete usare la stessa della vostra chiave primaria.

Una volta che il processo di criptazione è finito, reinserite il disco USB ed assicurativi che venga montato correttamente. Copiate interamente la cartella .gnupg nel disco criptato:

```
$ cp -a ~/.gnupg /media/disk/foo/gnupg-backup
```

Ora dovrete verificare che tutto continui a funzionare:

```
$ gpg --homedir=/media/disk/foo/gnupg-backup --list-key [fpr]
```

Se non vedete errori, allora dovrete avere fatto tutto con successo. Smontate il disco USB, etichettatelo per bene di modo da evitare di distruggerne il contenuto non appena vi serve una chiavetta USB a caso, ed infine mettetelo in un posto sicuro – ma non troppo lontano, perché vi servirà di tanto in tanto per modificare le identità, aggiungere o revocare sottochiavi, o firmare le chiavi di altre persone.

## Togliete la chiave primaria dalla vostra home

I file che si trovano nella vostra cartella home non sono poi così ben protetti come potreste pensare. Potrebbero essere letti o trafugati in diversi modi:

- accidentalmente quando fate una rapida copia della cartella home per configurare una nuova postazione
- da un amministratore di sistema negligente o malintenzionato
- attraverso copie di riserva insicure
- attraverso malware installato in alcune applicazioni (browser, lettori PDF, eccetera)
- attraverso coercizione quando attraversate confini internazionali

Proteggere la vostra chiave con una buona passphrase aiuta notevolmente a ridurre i rischi elencati qui sopra, ma le passphrase possono essere scoperte attraverso i keylogger, il shoulder-surfing, o altri modi. Per questi motivi, nella configurazione si raccomanda di rimuovere la chiave primaria dalla vostra cartella home e la si archivia su un dispositivo disconnesso.

**Warning:** Per favore, fate riferimento alla sezione precedente e assicuratevi di aver fatto una copia di riserva totale della cartella GnuPG. Quello che stiamo per fare renderà la vostra chiave inutile se non avete delle copie di riserva utilizzabili!

Per prima cosa, identificate il keygrip della vostra chiave primaria:

```
$ gpg --with-keygrip --list-key [fpr]
```

L'output assomiglierà a questo:

Trovate la voce keygrid che si trova sotto alla riga pub (appena sotto all'impronta digitale della chiave primaria). Questo corrisponderà direttamente ad un file nella cartella `~/.gnupg`:

Quello che dovrete fare è rimuovere il file .key che corrisponde al keygrip della chiave primaria:

Ora, se eseguite il comando `--list-secret-keys`, vedrete che la chiave primaria non compare più (il simbolo # indica che non è disponibile):

```
$ gpg --list-secret-keys
sec# rsa2048 2018-01-24 [SC] [expires: 2020-01-24]
          0000000000000000000000000000AAAABBBBCCCCDDDD
uid          [ultimate] Alice Dev <adev@kernel.org>
ssb    rsa2048 2018-01-24 [E] [expires: 2020-01-24]
ssb    ed25519 2018-01-24 [S]
```

Dovreste rimuovere anche i file `secring.gpg` che si trovano nella cartella `~/.gnupg`, in quanto rimasugli delle versioni precedenti di GnuPG.

## Se non avete la cartella “private-keys-v1.d”

Se non avete la cartella `~/.gnupg/private-keys-v1.d`, allora le vostre chiavi segrete sono ancora salvate nel vecchio file `secring.gpg` usato da GnuPG v1. Effettuare una qualsiasi modifica alla vostra chiave, come cambiare la passphrase o aggiungere una sottochiave, dovrebbe convertire automaticamente il vecchio formato `secring.gpg` nel nuovo `private-keys-v1.d`.

Una volta che l'avete fatto, assicuratevi di rimuovere il file `secring.gpg`, che continua a contenere la vostra chiave privata.

## Spostare le sottochiavi in un apposito dispositivo criptato

Nonostante la chiave primaria sia ora al riparo da occhi e mani indiscrete, le sottochiavi si trovano ancora nella vostra cartella `home`. Chiunque riesca a mettere le sue mani su quelle chiavi riuscirà a decriptare le vostre comunicazioni o a falsificare le vostre firme (se conoscono la passphrase). Inoltre, ogni volta che viene fatta un'operazione con GnuPG, le chiavi vengono caricate nella memoria di sistema e potrebbero essere rubate con l'uso di malware sofisticati (pensate a Meltdown e a Spectre).

Il miglior modo per proteggere le proprie chiavi è di spostarle su un dispositivo specializzato in grado di effettuare operazioni smartcard.

## I benefici di una smartcard

Una smartcard contiene un chip crittografico che è capace di immagazzinare le chiavi private ed effettuare operazioni crittografiche direttamente sulla carta stessa. Dato che la chiave non lascia mai la smartcard, il sistema operativo usato sul computer non sarà in grado di accedere alle chiavi. Questo è molto diverso dai dischi USB criptati che abbiamo usato allo scopo di avere una copia di riserva sicura – quando il dispositivo USB è connesso e montato, il sistema operativo potrà accedere al contenuto delle chiavi private.

L'uso di un disco USB criptato non può sostituire le funzioni di un dispositivo capace di operazioni di tipo smartcard.

## Dispositivi smartcard disponibili

A meno che tutti i vostri computer dispongano di lettori smartcard, il modo più semplice è equipaggiarsi di un dispositivo USB specializzato che implementi le funzionalità delle smartcard. Sul mercato ci sono diverse soluzioni disponibili:

- [Nitrokey Start](#): è Open hardware e Free Software, è basata sul progetto [GnuK](#) della FSij. Questo è uno dei pochi dispositivi a supportare le chiavi ECC ED25519, ma offre meno funzionalità di sicurezza (come la resistenza alla manomissione o alcuni attacchi ad un canale laterale).
- [Nitrokey Pro 2](#): è simile alla Nitrokey Start, ma è più resistente alla manomissione e offre più funzionalità di sicurezza. La Pro 2 supporta la crittografia ECC (NISTP).
- [Yubikey 5](#): l'hardware e il software sono proprietari, ma è più economica della Nitrokey Pro ed è venduta anche con porta USB-C il che è utile con i computer portatili più recenti.

In aggiunta, offre altre funzionalità di sicurezza come FIDO, U2F, e ora supporta anche le chiavi ECC (NISTP)

Su LWN c'è una buona recensione dei modelli elencati qui sopra e altri. La scelta dipenderà dal costo, dalla disponibilità nella vostra area geografica e vostre considerazioni sull'hardware aperto/proprietario.

Se volete usare chiavi ECC, la vostra migliore scelta sul mercato è la Nitrokey Start.

### Configurare il vostro dispositivo smartcard

Il vostro dispositivo smartcard dovrebbe iniziare a funzionare non appena lo collegate ad un qualsiasi computer Linux moderno. Potete verificarlo eseguendo:

```
$ gpg --card-status
```

Se vedete tutti i dettagli della smartcard, allora ci siamo. Sfortunatamente, affrontare tutti i possibili motivi per cui le cose potrebbero non funzionare non è lo scopo di questa guida. Se avete problemi nel far funzionare la carta con GnuPG, cercate aiuto attraverso i soliti canali di supporto.

Per configurare la vostra smartcard, dato che non c'è una via facile dalla riga di comando, dovete usare il menu di GnuPG:

```
$ gpg --card-edit  
[...omitted...]  
gpg/card> admin  
Admin commands are allowed  
gpg/card> passwd
```

Dovreste impostare il PIN dell'utente (1), quello dell'amministratore (3) e il codice di reset (4). Assicuratevi di annotare e salvare questi codici in un posto sicuro - specialmente il PIN dell'amministratore e il codice di reset (che vi permetterà di azzerare completamente la smartcard). Il PIN dell'amministratore viene usato così raramente che è inevitabile dimenticarselo se non lo si annota.

Tornando al nostro menu, potete impostare anche altri valori (come il nome, il sesso, informazioni d'accesso, eccetera), ma non sono necessari e aggiunge altre informazioni sulla carta che potrebbero trapelare in caso di smarrimento.

---

**Note:** A dispetto del nome "PIN", né il PIN utente né quello dell'amministratore devono essere esclusivamente numerici.

---

## Spostare le sottochiavi sulla smartcard

Uscite dal menu (usando “q”) e salverete tutte le modifiche. Poi, spostiamo tutte le sottochiavi sulla smartcard. Per la maggior parte delle operazioni vi serviranno sia la passphrase della chiave PGP che il PIN dell’amministratore:

```
$ gpg --edit-key [fpr]
Secret subkeys are available.

pub rsa2048/AAAABBBBCCCCDDDD
    created: 2018-01-23 expires: 2020-01-23 usage: SC
    trust: ultimate validity: ultimate
ssb rsa2048/1111222233334444
    created: 2018-01-23 expires: never usage: E
ssb ed25519/5555666677778888
    created: 2017-12-07 expires: never usage: S
[ultimate] (1). Alice Dev <adev@kernel.org>

gpg>
```

Usando `--edit-key` si tornerà alla modalità menu e noterete che la lista delle chiavi è leggermente diversa. Da questo momento in poi, tutti i comandi saranno eseguiti nella modalità menu, come indicato da `gpg>`.

Per prima cosa, selezioniamo la chiave che verrà messa sulla carta – potete farlo digitando `key 1` (è la prima della lista, la sottochiave **[E]**):

```
gpg> key 1
```

Nell’output dovreste vedere `ssb*` associato alla chiave **[E]**. Il simbolo \* indica che la chiave è stata “selezionata”. Funziona come un interruttore, ovvero se scrivete nuovamente `key 1`, il simbolo \* sparirà e la chiave non sarà più selezionata.

Ora, spostiamo la chiave sulla smartcard:

```
gpg> keytocard
Please select where to store the key:
(2) Encryption key
Your selection? 2
```

Dato che è la nostra chiave **[E]**, ha senso metterla nella sezione criptata. Quando confermerete la selezione, vi verrà chiesta la passphrase della vostra chiave PGP, e poi il PIN dell’amministratore. Se il comando ritorna senza errori, allora la vostra chiave è stata spostata con successo.

**Importante:** digitate nuovamente `key 1` per deselectare la prima chiave e selezionate la seconda chiave **[S]** con `key 2`:

```
gpg> key 1
gpg> key 2
gpg> keytocard
Please select where to store the key:
```

```
(1) Signature key  
(3) Authentication key  
Your selection? 1
```

Potete usare la chiave **[S]** sia per firmare che per autenticare, ma vogliamo che sia nella sezione di firma, quindi scegliete (1). Ancora una volta, se il comando ritorna senza errori, allora l'operazione è avvenuta con successo:

```
gpg> q  
Save changes? (y/N) y
```

Salvando le modifiche cancellerete dalla vostra cartella home tutte le chiavi che avete spostato sulla carta (ma questo non è un problema, perché abbiamo fatto delle copie di sicurezza nel caso in cui dovessimo configurare una nuova smartcard).

### Verificare che le chiavi siano state spostate

Ora, se dovreste usare l'opzione `--list-secret-keys`, vedrete una sottile differenza nell'output:

```
$ gpg --list-secret-keys  
sec# rsa2048 2018-01-24 [SC] [expires: 2020-01-24]  
          00000000000000000000000000000000AAAAABBBBCCCCDDDD  
uid          [ultimate] Alice Dev <audev@kernel.org>  
ssb> rsa2048 2018-01-24 [E] [expires: 2020-01-24]  
ssb> ed25519 2018-01-24 [S]
```

Il simbolo `>` in `ssb>` indica che la sottochiave è disponibile solo nella smartcard. Se tornate nella vostra cartella delle chiavi segrete e guardate al suo contenuto, noterete che i file `.key` sono stati sostituiti con degli stub:

```
$ cd ~/.gnupg/private-keys-v1.d  
$ strings *.key | grep 'private-key'
```

Per indicare che i file sono solo degli stub e che in realtà il contenuto è sulla smartcard, l'output dovrebbe mostrarvi `shadowed-private-key`.

### Verificare che la smartcard funzioni

Per verificare che la smartcard funzioni come dovuto, potete creare una firma:

```
$ echo "Hello world" | gpg --clearsign > /tmp/test.asc  
$ gpg --verify /tmp/test.asc
```

Col primo comando dovrebbe chiedervi il PIN della smartcard, e poi dovrebbe mostrare “Good signature” dopo l'esecuzione di `gpg --verify`.

Complimenti, siete riusciti a rendere estremamente difficile il furto della vostra identità digitale di sviluppatore.

## Altre operazioni possibili con GnuPG

Segue un breve accenno ad alcune delle operazioni più comuni che dovrete fare con le vostre chiavi PGP.

### Montare il disco con la chiave primaria

Vi servirà la vostra chiave principale per tutte le operazioni che seguiranno, per cui per prima cosa dovrete accedere ai vostri backup e dire a GnuPG di usarli:

```
$ export GNUPGHOME=/media/disk/foo/gnupg-backup  
$ gpg --list-secret-keys
```

Dovete assicurarvi di vedere `sec` e non `sec#` nell'output del programma (il simbolo `#` significa che la chiave non è disponibile e che state ancora utilizzando la vostra solita cartella di lavoro).

### Estendere la data di scadenza di una chiave

La chiave principale ha una data di scadenza di 2 anni dal momento della sua creazione. Questo per motivi di sicurezza e per rendere obsolete le chiavi che, eventualmente, dovessero sparire dai keyserver.

Per estendere di un anno, dalla data odierna, la scadenza di una vostra chiave, eseguite:

```
$ gpg --quick-set-expire [fpr] 1y
```

Se per voi è più facile da memorizzare, potete anche utilizzare una data specifica (per esempio, il vostro compleanno o capodanno):

```
$ gpg --quick-set-expire [fpr] 2020-07-01
```

Ricordatevi di inviare l'aggiornamento ai keyserver:

```
$ gpg --send-key [fpr]
```

### Aggiornare la vostra cartella di lavoro dopo ogni modifica

Dopo aver fatto delle modifiche alle vostre chiavi usando uno spazio a parte, dovreste importarle nella vostra cartella di lavoro abituale:

```
$ gpg --export | gpg --homedir ~/.gnupg --import  
$ unset GNUPGHOME
```

### Usare PGP con Git

Una delle caratteristiche fondanti di Git è la sua natura decentralizzata – una volta che il repository è stato clonato sul vostro sistema, avete la storia completa del progetto, inclusi i suoi tag, i commit ed i rami. Tuttavia, con i centinaia di repository clonati che ci sono in giro, come si fa a verificare che la loro copia di `linux.git` non è stata manomessa da qualcuno?

Oppure, cosa succede se viene scoperta una backdoor nel codice e la riga “Autore” dice che sei stato tu, mentre tu sei abbastanza sicuro di [non averci niente a che fare](#)?

Per risolvere entrambi i problemi, Git ha introdotto l’integrazione con PGP. I tag firmati dimostrano che il repository è integro assicurando che il suo contenuto è lo stesso che si trova sulle macchine degli sviluppatori che hanno creato il tag; mentre i commit firmati rendono praticamente impossibile ad un malintenzionato di impersonarvi senza avere accesso alle vostre chiavi PGP.

### Configurare git per usare la vostra chiave PGP

Se avete solo una chiave segreta nel vostro portachiavi, allora non avete nulla da fare in più dato che sarà la vostra chiave di base. Tuttavia, se dovreste avere più chiavi segrete, potete dire a git quale dovrebbe usare ([fpr] è la vostra impronta digitale):

```
$ git config --global user.signingKey [fpr]
```

**IMPORTANTE:** se avete una comando dedicato per `gpg2`, allora dovreste dire a git di usare sempre quello piuttosto che il vecchio comando `gpg`:

```
$ git config --global gpg.program gpg2
```

### Come firmare i tag

Per creare un tag firmato, passate l’opzione `-s` al comando `tag`:

```
$ git tag -s [tagname]
```

La nostra raccomandazione è quella di firmare sempre i tag git, perché questo permette agli altri sviluppatori di verificare che il repository git dal quale stanno prendendo il codice non è stato alterato intenzionalmente.

### Come verificare i tag firmati

Per verificare un tag firmato, potete usare il comando `verify-tag`:

```
$ git verify-tag [tagname]
```

Se state prendendo un tag da un fork del repository del progetto, git dovrebbe verificare automaticamente la firma di quello che state prendendo e vi mostrerà il risultato durante l’operazione di merge:

```
$ git pull [url] tags/sometag
```

Il merge conterrà qualcosa di simile:

```
Merge tag 'sometag' of [url]

[Tag message]

# gpg: Signature made [...]
# gpg: Good signature from [...]
```

Se state verificando il tag di qualcun altro, allora dovete importare la loro chiave PGP. Fate riferimento alla sezione “[Come verificare l’identità degli sviluppatori del kernel](#)” che troverete più avanti.

## Configurare git per firmare sempre i tag con annotazione

Se state creando un tag con annotazione è molto probabile che vogliate firmarlo. Per imporre a git di firmare sempre un tag con annotazione, dovete impostare la seguente opzione globale:

```
$ git config --global tag.forceSignAnnotated true
```

## Come usare commit firmati

Creare dei commit firmati è facile, ma è molto più difficile utilizzarli nello sviluppo del kernel linux per via del fatto che ci si affida alle liste di discussione e questo modo di procedere non mantiene le firme PGP nei commit. In aggiunta, quando si usa *rebase* nel proprio repository locale per allinearsi al kernel anche le proprie firme PGP verranno scartate. Per questo motivo, la maggior parte degli sviluppatori del kernel non si preoccupano troppo di firmare i propri commit ed ignoreranno quelli firmati che si trovano in altri repository usati per il proprio lavoro.

Tuttavia, se avete il vostro repository di lavoro disponibile al pubblico su un qualche servizio di hosting git (kernel.org, infradead.org, ozlabs.org, o altri), allora la raccomandazione è di firmare tutti i vostri commit anche se gli sviluppatori non ne beneficeranno direttamente.

Vi raccomandiamo di farlo per i seguenti motivi:

1. Se dovesse mai esserci la necessità di fare delle analisi forensi o tracciare la provenienza di un codice, anche sorgenti mantenuti esternamente che hanno firme PGP sui commit avranno un certo valore a questo scopo.
2. Se dovesse mai capitarevi di clonare il vostro repository locale (per esempio dopo un danneggiamento del disco), la firma vi permetterà di verificare l’integrità del repository prima di riprendere il lavoro.
3. Se qualcuno volesse usare *cherry-pick* sui vostri commit, allora la firma permetterà di verificare l’integrità dei commit prima di applicarli.

### Creare commit firmati

Per creare un commit firmato, dovete solamente aggiungere l'opzione `-S` al comando `git commit` (si usa la lettera maiuscola per evitare conflitti con un'altra opzione):

```
$ git commit -S
```

### Configurare git per firmare sempre i commit

Potete dire a git di firmare sempre i commit:

```
git config --global commit.gpgSign true
```

---

**Note:** Assicuratevi di aver configurato `gpg-agent` prima di abilitare questa opzione.

---

### Come verificare l'identità degli sviluppatori del kernel

Firmare i tag e i commit è facile, ma come si fa a verificare che la chiave usata per firmare qualcosa appartenga davvero allo sviluppatore e non ad un impostore?

### Configurare l'auto-key-retrieval usando WKD e DANE

Se non siete ancora in possesso di una vasta collezione di chiavi pubbliche di altri sviluppatori, allora potreste iniziare il vostro portachiavi affidandovi ai servizi di auto-scoperta e auto-recupero. GnuPG può affidarsi ad altre tecnologie di delega della fiducia, come DNSSEC e TLS, per sostenervi nel caso in cui iniziare una propria rete di fiducia da zero sia troppo scoraggiante.

Aggiungete il seguente testo al vostro file `~/.gnupg/gpg.conf`:

```
auto-key-locate wkd,dane,local  
auto-key-retrieve
```

La *DNS-Based Authentication of Named Entities* (“DANE”) è un metodo per la pubblicazione di chiavi pubbliche su DNS e per renderle sicure usando zone firmate con DNSSEC. Il *Web Key Directory* (“WKD”) è un metodo alternativo che usa https a scopo di ricerca. Quando si usano DANE o WKD per la ricerca di chiavi pubbliche, GnuPG validerà i certificati DNSSEC o TLS prima di aggiungere al vostro portachiavi locale le eventuali chiavi trovate.

Kernel.org pubblica la WKD per tutti gli sviluppatori che hanno un account kernel.org. Una volta che avete applicato le modifiche al file `gpg.conf`, potrete auto-recuperare le chiavi di Linus Torvalds e Greg Kroah-Hartman (se non le avete già):

```
$ gpg --locate-keys torvalds@kernel.org gregkh@kernel.org
```

Se avete un account kernel.org, al fine di rendere più utile l'uso di WKD da parte di altri sviluppatori del kernel, dovreste [aggiungere alla vostra chiave lo UID di kernel.org](#).

## Web of Trust (WOT) o Trust on First Use (TOFU)

PGP incorpora un meccanismo di delega della fiducia conosciuto come “Web of Trust”. Di base, questo è un tentativo di sostituire la necessità di un’autorità certificativa centralizzata tipica del mondo HTTPS/TLS. Invece di avere svariati produttori software che decidono chi dovrebbero essere le entità di certificazione di cui dovreste fidarvi, PGP lascia la responsabilità ad ogni singolo utente.

Sfortunatamente, solo poche persone capiscono come funziona la rete di fiducia. Nonostante sia un importante aspetto della specifica OpenPGP, recentemente le versioni di GnuPG (2.2 e successive) hanno implementato un meccanismo alternativo chiamato “Trust on First Use” (TOFU). Potete pensare a TOFU come “ad un approccio all’fiducia simile ad SSH”. In SSH, la prima volta che vi connettete ad un sistema remoto, l’impronta digitale della chiave viene registrata e ricordata. Se la chiave dovesse cambiare in futuro, il programma SSH vi avviserà e si rifiuterà di connettersi, obbligandovi a prendere una decisione circa la fiducia che riponete nella nuova chiave. In modo simile, la prima volta che importate la chiave PGP di qualcuno, si assume sia valida. Se ad un certo punto GnuPG trova un’altra chiave con la stessa identità, entrambe, la vecchia e la nuova, verranno segnate come invalide e dovrete verificare manualmente quale tenere.

Vi raccomandiamo di usare il meccanismo TOFU+PGP (che è la nuova configurazione di base di GnuPG v2). Per farlo, aggiungete (o modificate) l’impostazione `trust-model` in `~/.gnupg/gpg.conf`:

```
trust-model tofu+pgp
```

## Come usare i keyserver in sicurezza

Se ottenete l’errore “No public key” quando cercate di validate il tag di qualcuno, allora dovreste cercare quella chiave usando un keyserver. È importante tenere bene a mente che non c’è alcuna garanzia che la chiave che avete recuperato da un keyserver PGP appartenga davvero alla persona reale - è progettato così. Dovreste usare il Web of Trust per assicurarvi che la chiave sia valida.

Come mantenere il Web of Trust va oltre gli scopi di questo documento, semplicemente perché farlo come si deve richiede sia sforzi che perseveranza che tendono ad andare oltre al livello di interesse della maggior parte degli esseri umani. Qui di seguito alcuni rapidi suggerimenti per aiutarvi a ridurre il rischio di importare chiavi maligne.

Primo, diciamo che avete provato ad eseguire `git verify-tag` ma restituisce un errore dicendo che la chiave non è stata trovata:

```
$ git verify-tag sunxi-fixes-for-4.15-2
gpg: Signature made Sun 07 Jan 2018 10:51:55 PM EST
gpg:                               using RSA key DA73759BF8619E484E5A3B47389A54219C0F2430
gpg:                               issuer "wens@...org"
gpg: Can't check signature: No public key
```

Cerchiamo nel keyserver per maggiori informazioni sull’impronta digitale della chiave (l’impronta digitale, probabilmente, appartiene ad una sottochiave, dunque non possiamo usarla direttamente senza trovare prima l’ID della chiave primaria associata ad essa):

```
$ gpg --search DA73759BF8619E484E5A3B47389A54219C0F2430
gpg: data source: hkp://keys.gnupg.net
(1) Chen-Yu Tsai <wens@...org>
    4096 bit RSA key C94035C21B4F2AEB, created: 2017-03-14, expires: 2019-03-
→15
Keys 1-1 of 1 for "DA73759BF8619E484E5A3B47389A54219C0F2430". Enter number(s),
→ N)ext, or Q)uit > q
```

Localizzate l'ID della chiave primaria, nel nostro esempio C94035C21B4F2AEB. Ora visualizzate le chiavi di Linus Torvalds che avete nel vostro portachiavi:

```
$ gpg --list-key torvalds@kernel.org
pub    rsa2048 2011-09-20 [SC]
      ABAF11C65A2970B130ABE3C479BE3E4300411886
uid          [ unknown] Linus Torvalds <torvalds@kernel.org>
sub    rsa2048 2011-09-20 [E]
```

Poi, aprete il [PGP pathfinder](#). Nel campo “From”, incollate l'impronta digitale della chiave di Linus Torvalds che si vede nell'output qui sopra. Nel campo “to”, incollate il key-id della chiave sconosciuta che avete trovato con `gpg --search`, e poi verificare il risultato:

- [Finding paths to Linus](#)

Se trovate un paio di percorsi affidabili è un buon segno circa la validità della chiave. Ora, potete aggiungerla al vostro portachiavi dal keyserver:

```
$ gpg --recv-key C94035C21B4F2AEB
```

Questa procedura non è perfetta, e ovviamente state riponendo la vostra fiducia nell'amministratore del servizio *PGP Pathfinder* sperando che non sia malintenzionato (infatti, questo va contro [Fidatevi degli sviluppatori e non dell'infrastruttura](#)). Tuttavia, se mantenete con cura la vostra rete di fiducia sarà un deciso miglioramento rispetto alla cieca fiducia nei keyserver.

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** ../../process/email-clients

**Translator** Alessia Mantegazza <[amantegazza@vaga.pv.it](mailto:amantegazza@vaga.pv.it)>

## Informazioni sui programmi di posta elettronica per Linux

### Git

Oggi giorno, la maggior parte degli sviluppatori utilizza `git send-email` al posto dei classici programmi di posta elettronica. Le pagine man sono abbastanza buone. Dal lato del ricevente, i manutentori utilizzano `git am` per applicare le patch.

Se siete dei novelli utilizzatori di git allora inviate la patch a voi stessi. Salvatela come testo includendo tutte le intestazioni. Poi eseguite il comando `git am messaggio-formato-testo.txt` e revisionatene il risultato con `git log`. Quando tutto funziona correttamente, allora potete inviare la patch alla lista di discussione più appropriata.

### Panoramica delle opzioni

Le patch per il kernel vengono inviate per posta elettronica, preferibilmente come testo integrante del messaggio. Alcuni manutentori accettano gli allegati, ma in questo caso gli allegati devono avere il *content-type* impostato come `text/plain`. Tuttavia, generalmente gli allegati non sono ben apprezzati perché rende più difficile citare porzioni di patch durante il processo di revisione.

Inoltre, è vivamente raccomandato l'uso di puro testo nel corpo del messaggio, sia per la patch che per qualsiasi altro messaggio. Il sito <https://useplaintext.email/> può esservi d'aiuto per configurare il vostro programma di posta elettronica.

I programmi di posta elettronica che vengono usati per inviare le patch per il kernel Linux dovrebbero inviarle senza alterazioni. Per esempio, non dovrebbero modificare o rimuovere tabulazioni o spazi, nemmeno all'inizio o alla fine delle righe.

Non inviate patch con `format=flowed`. Questo potrebbe introdurre interruzioni di riga inaspettate e indesiderate.

Non lasciate che il vostro programma di posta vada a capo automaticamente. Questo può corrompere le patch.

I programmi di posta non dovrebbero modificare la codifica dei caratteri nel testo. Le patch inviate per posta elettronica dovrebbero essere codificate in ASCII o UTF-8. Se configurate il vostro programma per inviare messaggi codificati con UTF-8 eviterete possibili problemi di codifica.

I programmi di posta dovrebbero generare e mantenere le intestazioni "References" o "In-Reply-To:" cosicché la discussione non venga interrotta.

Di solito, il copia-e-incolla (o taglia-e-incolla) non funziona con le patch perché le tabulazioni vengono convertite in spazi. Usando xclipboard, xclip e/o xcutsel potrebbe funzionare, ma è meglio che lo verifichiate o meglio ancora: non usate il copia-e-incolla.

Non usate firme PGP/GPG nei messaggi che contengono delle patch. Questo impedisce il corretto funzionamento di alcuni script per leggere o applicare patch (questo si dovrebbe poter correggere).

Prima di inviare le patch sulle liste di discussione Linux, può essere una buona idea quella di inviare la patch a voi stessi, salvare il messaggio ricevuto, e applicarlo ai sorgenti con successo.

### Alcuni suggerimenti per i programmi di posta elettronica (MUA)

Qui troverete alcuni suggerimenti per configurare i vostri MUA allo scopo di modificare ed inviare patch per il kernel Linux. Tuttavia, questi suggerimenti non sono da considerarsi come un riassunto di una configurazione completa.

Legenda:

- TUI = interfaccia utente testuale (*text-based user interface*)
- GUI = interfaccia utente grafica (*graphical user interface*)

### Alpine (TUI)

Opzioni per la configurazione:

Nella sezione *Sending Preferences*:

- *Do Not Send Flowed Text* deve essere enabled
- *Strip Whitespace Before Sending* deve essere disabled

Quando state scrivendo un messaggio, il cursore dev'essere posizionato dove volete che la patch inizi, poi premendo CTRL-R vi verrà chiesto di selezionare il file patch da inserire nel messaggio.

### Claws Mail (GUI)

Funziona. Alcune persone riescono ad usarlo con successo per inviare le patch.

Per inserire una patch usate *Messaggio→Inserisci file* (CTRL-I) oppure un editor esterno.

Se la patch che avete inserito dev'essere modificata usato la finestra di scrittura di Claws, allora assicuratevi che l'"auto-interruzione" sia disabilitata *Configurazione→Preferenze→Composizione→Interruzione riga*.

### Evolution (GUI)

Alcune persone riescono ad usarlo con successo per inviare le patch.

**Quando state scrivendo una lettera selezionate: Preformattato** da *Formato→Stile del paragrafo→Preformattato* (CTRL-7) o dalla barra degli strumenti

Poi per inserire la patch usate: *Inserisci→File di testo...* (ALT-N x)

Potete anche eseguire `diff -Nru old.c new.c | xclip`, selezionare *Preformattato*, e poi usare il tasto centrale del mouse.

## Kmail (GUI)

Alcune persone riescono ad usarlo con successo per inviare le patch.

La configurazione base che disabilita la composizione di messaggi HTML è corretta; non abilitatela.

Quando state scrivendo un messaggio, nel menu opzioni, togliete la selezione a "A capo automatico". L'unico svantaggio sarà che qualsiasi altra cosa scriviate nel messaggio non verrà mandata a capo in automatico ma dovrete farlo voi. Il modo più semplice per ovviare a questo problema è quello di scrivere il messaggio con l'opzione abilitata e poi di salvarlo nelle bozze. Riaprendo ora il messaggio dalle bozze le andate a capo saranno parte integrante del messaggio, per cui togliendo l'opzione "A capo automatico" non perderete nulla.

Alla fine del vostro messaggio, appena prima di inserire la vostra patch, aggiungete il delimitatore di patch: tre trattini (---).

Ora, dal menu *Messaggio*, selezionate *Inserisci file di testo...* quindi scegliete la vostra patch. Come soluzione aggiuntiva potreste personalizzare la vostra barra degli strumenti aggiungendo un'icona per *Inserisci file di testo*....

Allargate la finestra di scrittura abbastanza da evitare andate a capo. Questo perché in Kmail 1.13.5 (KDE 4.5.4), Kmail aggiunge andate a capo automaticamente al momento dell'invio per tutte quelle righe che graficamente, nella vostra finestra di composizione, si sono estese su una riga successiva. Disabilitare l'andata a capo automatica non è sufficiente. Dunque, se la vostra patch contiene delle righe molto lunghe, allora dovete allargare la finestra di composizione per evitare che quelle righe vadano a capo. Vedere: [https://bugs.kde.org/show\\_bug.cgi?id=174034](https://bugs.kde.org/show_bug.cgi?id=174034)

Potete firmare gli allegati con GPG, ma per le patch si preferisce aggiungerle al testo del messaggio per cui non usate la firma GPG. Firmare le patch inserite come testo del messaggio le rende più difficili da estrarre dalla loro codifica a 7-bit.

Se dovete assolutamente inviare delle patch come allegati invece di integrarle nel testo del messaggio, allora premete il tasto destro sull'allegato e selezionate *Proprietà*, e poi attivate *Suggerisci visualizzazione automatica* per far sì che l'allegato sia più leggibile venendo visualizzato come parte del messaggio.

Per salvare le patch inviate come parte di un messaggio, selezionate il messaggio che la contiene, premete il tasto destro e selezionate *Salva come*. Se il messaggio fu ben preparato, allora potrete usarlo interamente senza alcuna modifica. I messaggi vengono salvati con permessi di lettura-scrittura solo per l'utente, nel caso in cui vogliate copiarli altrove per renderli disponibili ad altri gruppi o al mondo, ricordatevi di usare chmod per cambiare i permessi.

## Lotus Notes (GUI)

Scappate finché potete.

### IBM Verse (Web GUI)

Vedi il commento per Lotus Notes.

### Mutt (TUI)

Un sacco di sviluppatori Linux usano mutt, per cui deve funzionare abbastanza bene.

Mutt non ha un proprio editor, quindi qualunque sia il vostro editor dovete configurarlo per non aggiungere automaticamente le andate a capo. Molti editor hanno un'opzione *Inserisci file* che inserisce il contenuto di un file senza alterarlo.

Per usare vim come editor per mutt:

```
set editor="vi"
```

Se per inserire la patch nel messaggio usate xclip, scrivete il comando:

```
:set paste
```

prima di premere il tasto centrale o shift-insert. Oppure usate il comando:

```
:r filename
```

(a)llega funziona bene senza set paste

Potete generare le patch con git format-patch e usare Mutt per inviarle:

```
$ mutt -H 0001-some-bug-fix.patch
```

Opzioni per la configurazione:

Tutto dovrebbe funzionare già nella configurazione base. Tuttavia, è una buona idea quella di impostare send\_charset:

```
set send_charset="us-ascii:utf-8"
```

Mutt è molto personalizzabile. Qui di seguito trovate la configurazione minima per iniziare ad usare Mutt per inviare patch usando Gmail:

```
# .muttrc
# ====== IMAP ======
set imap_user = 'yourusername@gmail.com'
set imap_pass = 'yourpassword'
set spoolfile = imaps://imap.gmail.com/INBOX
set folder = imaps://imap.gmail.com/
set record="imaps://imap.gmail.com/[Gmail]/Sent Mail"
set postponed="imaps://imap.gmail.com/[Gmail]/Drafts"
set mbox="imaps://imap.gmail.com/[Gmail]/All Mail"

# ====== SMTP ======
set smtp_url = "smtp://username@smtp.gmail.com:587/"
set smtp_pass = $imap_pass
```

```

set ssl_force_tls = yes # Require encrypted connection

# ===== Composition =====
set editor = `echo \$EDITOR`
set edit_headers = yes # See the headers when editing
set charset = UTF-8 # value of $LANG; also fallback for send_charset
# Sender, email address, and sign-off line must match
unset use_domain # because joe@localhost is just embarrassing
set realname = "YOUR NAME"
set from = "username@gmail.com"
set use_from = yes

```

La documentazione di Mutt contiene molte più informazioni:

<https://gitlab.com/muttmua/mutt/-/wikis/UseCases/Gmail>

<http://www.mutt.org/doc/manual/>

## Pine (TUI)

Pine aveva alcuni problemi con gli spazi vuoti, ma questi dovrebbero essere stati risolti.

Se potete usate alpine (il successore di pine).

Opzioni di configurazione:

- Nelle versioni più recenti è necessario avere `quell-flowed-text`
- l'opzione `no-strip-whitespace-before-send` è necessaria

## Sylpheed (GUI)

- funziona bene per aggiungere testo in linea (o usando allegati)
- permette di utilizzare editor esterni
- è lento su cartelle grandi
- non farà l'autenticazione TSL SMTP su una connessione non SSL
- ha un utile righello nella finestra di scrittura
- la rubrica non comprende correttamente il nome da visualizzare e l'indirizzo associato

## Thunderbird (GUI)

Thunderbird è un clone di Outlook a cui piace maciullare il testo, ma esistono modi per impedirglielo.

- permettere l'uso di editor esterni: La cosa più semplice da fare con Thunderbird e le patch è quello di usare l'estensione “external editor” e di usare il vostro \$EDITOR preferito per leggere/includere patch nel vostro messaggio. Per farlo, scaricate ed installate l'estensione e aggiungete un bottone per chiamarla rapidamente usando *Visualizza→Barra*

*degli strumenti* → *Personalizza...*; una volta fatto potrete richiamarlo premendo sul bottone mentre siete nella finestra *Scrivi*

Tenete presente che “external editor” richiede che il vostro editor non faccia alcun fork, in altre parole, l’editor non deve ritornare prima di essere stato chiuso. Potreste dover passare dei parametri aggiuntivi al vostro editor oppure cambiargli la configurazione. Per esempio, usando gvim dovrete aggiungere l’opzione `-f /usr/bin/gvim -f` (Se il binario si trova in `/usr/bin`) nell’apposito campo nell’interfaccia di configurazione di *external editor*. Se usate altri editor consultate il loro manuale per sapere come configurarli.

Per rendere l’editor interno un po’ più sensato, fate così:

- Modificate le impostazioni di Thunderbird per far sì che non usi `format=flowed`. Andate in *Modifica* → *Preferenze* → *Avanzate* → *Editor di configurazione* per invocare il registro delle impostazioni.
- impostate `mailnews.send_plaintext_flowed` a `false`
- impostate `mailnews.wraplength` da 72 a 0
- *Visualizza* → *Corpo del messaggio come* → *Testo semplice*
- *Visualizza* → *Codifica del testo* → *Unicode*

### TkRat (GUI)

Funziona. Usare “Inserisci file...” o un editor esterno.

### Gmail (Web GUI)

Non funziona per inviare le patch.

Il programma web Gmail converte automaticamente i tab in spazi.

Allo stesso tempo aggiunge andata a capo ogni 78 caratteri. Comunque il problema della conversione fra spazi e tab può essere risolto usando un editor esterno.

Un altro problema è che Gmail usa la codifica base64 per tutti quei messaggi che contengono caratteri non ASCII. Questo include cose tipo i nomi europei.

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l’unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/kernel-enforcement-statement.rst

**Translator** Federico Vaga <[federico.vaga@vaga.pv.it](mailto:federico.vaga@vaga.pv.it)>

## Applicazione della licenza sul kernel Linux

Come sviluppatori del kernel Linux, abbiamo un certo interesse su come il nostro software viene usato e su come la sua licenza viene fatta rispettare. Il rispetto reciproco degli obblighi di condivisione della GPL-2.0 è fondamentale per la sostenibilità di lungo periodo del nostro software e della nostra comunità.

Benché ognuno abbia il diritto a far rispettare il diritto d'autore per i propri contributi alla nostra comunità, condividiamo l'interesse a far sì che ogni azione individuale nel far rispettare i propri diritti sia condotta in modo da portare beneficio alla comunità e che non abbia, involontariamente, impatti negativi sulla salute e la crescita del nostro ecosistema software. Al fine di scoraggiare l'esecuzione di azioni inutili, concordiamo che è nel migliore interesse della nostra comunità di sviluppo di impegnarci nel rispettare i seguenti obblighi nei confronti degli utenti del kernel Linux per conto nostro e di qualsiasi successore ai nostri interessi sul diritto d'autore:

Malgrado le clausole di risoluzione della licenza GPL-2.0, abbiamo concordato che è nel migliore interesse della nostra comunità di sviluppo adottare le seguenti disposizioni della GPL-3.0 come permessi aggiuntivi alla nostra licenza nei confronti di qualsiasi affermazione non difensiva di diritti sulla licenza.

In ogni caso, se cessano tutte le violazioni di questa Licenza, allora la tua licenza da parte di un dato detentore del copyright viene ripristinata (a) in via cautelativa, a meno che e fino a quando il detentore del copyright non cessa esplicitamente e definitivamente la tua licenza, e (b) in via permanente se il detentore del copyright non ti notifica in alcun modo la violazione entro 60 giorni dalla cessazione della licenza.

Inoltre, la tua licenza da parte di un dato detentore del copyright viene ripristinata in maniera permanente se il detentore del copyright ti notifica la violazione in maniera adeguata, se questa è la prima volta che ricevi una notifica di violazione di questa Licenza (per qualunque Programma) dallo stesso detentore di copyright, e se rimedi alla violazione entro 30 giorni dalla data di ricezione della notifica di violazione.

Fornendo queste garanzie, abbiamo l'intenzione di incoraggiare l'uso del software. Vogliamo che le aziende e le persone usino, modifichino e distribuiscano a questo software. Vogliamo lavorare con gli utenti in modo aperto e trasparente per eliminare ogni incertezza circa le nostre aspettative sul rispetto o l'ottemperanza alla licenza che possa limitare l'uso del nostro software. Vediamo l'azione legale come ultima spiaggia, da avviare solo quando gli altri sforzi della comunità hanno fallito nel risolvere il problema.

Per finire, una volta che un problema di non rispetto della licenza viene risolto, speriamo che gli utenti si sentano i benvenuti ad aggregarsi a noi nello sviluppo di questo progetto. Lavorando assieme, saremo più forti.

Ad eccezione deve specificato, parliamo per noi stessi, e non per una qualsiasi azienda per la quale lavoriamo oggi, o per cui abbiamo lavorato in passato, o lavoreremo in futuro.

- Laura Abbott
- Bjorn Andersson (Linaro)
- Andrea Arcangeli
- Neil Armstrong
- Jens Axboe

- Pablo Neira Ayuso
- Khalid Aziz
- Ralf Baechle
- Felipe Balbi
- Arnd Bergmann
- Ard Biesheuvel
- Tim Bird
- Paolo Bonzini
- Christian Borntraeger
- Mark Brown (Linaro)
- Paul Burton
- Javier Martinez Canillas
- Rob Clark
- Kees Cook (Google)
- Jonathan Corbet
- Dennis Dalessandro
- Vivien Didelot (Savoir-faire Linux)
- Hans de Goede
- Mel Gorman (SUSE)
- Sven Eckelmann
- Alex Elder (Linaro)
- Fabio Estevam
- Larry Finger
- Bhumika Goyal
- Andy Gross
- Juergen Gross
- Shawn Guo
- Ulf Hansson
- Stephen Hemminger (Microsoft)
- Tejun Heo
- Rob Herring
- Masami Hiramatsu
- Michal Hocko
- Simon Hormann

- Johan Hovold (Hovold Consulting AB)
- Christophe JAILLET
- Olof Johansson
- Lee Jones (Linaro)
- Heiner Kallweit
- Srinivas Kandagatla
- Jan Kara
- Shuah Khan (Samsung)
- David Kershner
- Jaeyeuk Kim
- Namhyung Kim
- Colin Ian King
- Jeff Kirsher
- Greg Kroah-Hartman (Linux Foundation)
- Christian König
- Vinod Koul
- Krzysztof Kozlowski
- Viresh Kumar
- Aneesh Kumar K.V
- Julia Lawall
- Doug Ledford
- Chuck Lever (Oracle)
- Daniel Lezcano
- Shaohua Li
- Xin Long
- Tony Luck
- Catalin Marinas (Arm Ltd)
- Mike Marshall
- Chris Mason
- Paul E. McKenney
- Arnaldo Carvalho de Melo
- David S. Miller
- Ingo Molnar
- Kuninori Morimoto

- Trond Myklebust
- Martin K. Petersen (Oracle)
- Borislav Petkov
- Jiri Pirko
- Josh Poimboeuf
- Sebastian Reichel (Collabora)
- Guenter Roeck
- Joerg Roedel
- Leon Romanovsky
- Steven Rostedt (VMware)
- Frank Rowand
- Ivan Safonov
- Anna Schumaker
- Jes Sorensen
- K.Y. Srinivasan
- David Sterba (SUSE)
- Heiko Stuebner
- Jiri Kosina (SUSE)
- Willy Tarreau
- Dmitry Torokhov
- Linus Torvalds
- Thierry Reding
- Rik van Riel
- Luis R. Rodriguez
- Geert Uytterhoeven (Glider bvba)
- Eduardo Valentin (Amazon.com)
- Daniel Vetter
- Linus Walleij
- Richard Weinberger
- Dan Williams
- Rafael J. Wysocki
- Arvind Yadav
- Masahiro Yamada
- Wei Yongjun

- Lv Zheng
- Marc Zyngier (Arm Ltd)

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/kernel-driver-statement.rst

**Translator** Federico Vaga <[federico.vaga@vaga.pv.it](mailto:federico.vaga@vaga.pv.it)>

## Dichiarazioni sui driver per il kernel

### Presa di posizione sui moduli per il kernel Linux

Noi, i sottoscritti sviluppatori del kernel, consideriamo pericoloso o indesiderato qualsiasi modulo o driver per il kernel Linux di tipo *a sorgenti chiusi* (*closed-source*). Ripetutamente, li abbiamo trovati deleteri per gli utenti Linux, le aziende, ed in generale l'ecosistema Linux. Questi moduli impediscono l'apertura, la stabilità, la flessibilità, e la manutenibilità del modello di sviluppo di Linux e impediscono ai loro utenti di beneficiare dell'esperienza dalla comunità Linux. I fornitori che distribuiscono codice a sorgenti chiusi obbligano i propri utenti a rinunciare ai principali vantaggi di Linux o a cercarsi nuovi fornitori. Perciò, al fine di sfruttare i vantaggi che codice aperto ha da offrire, come l'abbattimento dei costi e un supporto condìvisio, spingiamo i fornitori ad adottare una politica di supporto ai loro clienti Linux che preveda il rilascio dei sorgenti per il kernel.

Parliamo solo per noi stessi, e non per una qualsiasi azienda per la quale lavoriamo oggi, o abbiamo lavorato in passato, o lavoreremo in futuro.

- Dave Airlie
- Nick Andrew
- Jens Axboe
- Ralf Baechle
- Felipe Balbi
- Ohad Ben-Cohen
- Muli Ben-Yehuda
- Jiri Benc
- Arnd Bergmann
- Thomas Bogendoerfer
- Vitaly Bordug
- James Bottomley
- Josh Boyer
- Neil Brown

- Mark Brown
- David Brownell
- Michael Buesch
- Franck Bui-Huu
- Adrian Bunk
- François Cami
- Ralph Campbell
- Luiz Fernando N. Capitulino
- Mauro Carvalho Chehab
- Denis Cheng
- Jonathan Corbet
- Glauber Costa
- Alan Cox
- Magnus Damm
- Ahmed S. Darwish
- Robert P. J. Day
- Hans de Goede
- Arnaldo Carvalho de Melo
- Helge Deller
- Jean Delvare
- Mathieu Desnoyers
- Sven-Thorsten Dietrich
- Alexey Dobriyan
- Daniel Drake
- Alex Dubov
- Randy Dunlap
- Michael Ellerman
- Pekka Enberg
- Jan Engelhardt
- Mark Fasheh
- J. Bruce Fields
- Larry Finger
- Jeremy Fitzhardinge
- Mike Frysinger

- Kumar Gala
- Robin Getz
- Liam Girdwood
- Jan-Benedict Glaw
- Thomas Gleixner
- Brice Goglin
- Cyrill Gorcunov
- Andy Gospodarek
- Thomas Graf
- Krzysztof Halasa
- Harvey Harrison
- Stephen Hemminger
- Michael Hennerich
- Tejun Heo
- Benjamin Herrenschmidt
- Kristian Høgsberg
- Henrique de Moraes Holschuh
- Marcel Holtmann
- Mike Isely
- Takashi Iwai
- Olof Johansson
- Dave Jones
- Jesper Juhl
- Matthias Kaehlcke
- Kenji Kaneshige
- Jan Kara
- Jeremy Kerr
- Russell King
- Olaf Kirch
- Roel Kluin
- Hans-Jürgen Koch
- Auke Kok
- Peter Korsgaard
- Jiri Kosina

- Aaro Koskinen
- Mariusz Kozlowski
- Greg Kroah-Hartman
- Michael Krufky
- Aneesh Kumar
- Clemens Ladisch
- Christoph Lameter
- Gunnar Larisch
- Anders Larsen
- Grant Likely
- John W. Linville
- Yinghai Lu
- Tony Luck
- Pavel Macheck
- Matt Mackall
- Paul Mackerras
- Roland McGrath
- Patrick McHardy
- Kyle McMartin
- Paul Menage
- Thierry Merle
- Eric Miao
- Akinobu Mita
- Ingo Molnar
- James Morris
- Andrew Morton
- Paul Mundt
- Oleg Nesterov
- Luca Olivetti
- S.Çağlar Onur
- Pierre Ossman
- Keith Owens
- Venkatesh Pallipadi
- Nick Piggin

- Nicolas Pitre
- Evgeniy Polyakov
- Richard Purdie
- Mike Rapoport
- Sam Ravnborg
- Gerrit Renker
- Stefan Richter
- David Rientjes
- Luis R. Rodriguez
- Stefan Roese
- Francois Romieu
- Rami Rosen
- Stephen Rothwell
- Maciej W. Rozycki
- Mark Salyzyn
- Yoshinori Sato
- Deepak Saxena
- Holger Schurig
- Amit Shah
- Yoshihiro Shimoda
- Sergei Shtylyov
- Kay Sievers
- Sebastian Siewior
- Rik Snel
- Jes Sorensen
- Alexey Starikovskiy
- Alan Stern
- Timur Tabi
- Hirokazu Takata
- Eliezer Tamir
- Eugene Teo
- Doug Thompson
- FUJITA Tomonori
- Dmitry Torokhov

- Marcelo Tosatti
- Steven Toth
- Theodore Tso
- Matthias Urlichs
- Geert Uytterhoeven
- Arjan van de Ven
- Ivo van Doorn
- Rik van Riel
- Wim Van Sebroeck
- Hans Verkuil
- Horst H. von Brand
- Dmitri Vorobiev
- Anton Vorontsov
- Daniel Walker
- Johannes Weiner
- Harald Welte
- Matthew Wilcox
- Dan J. Williams
- Darrick J. Wong
- David Woodhouse
- Chris Wright
- Bryan Wu
- Rafael J. Wysocki
- Herbert Xu
- Vlad Yasevich
- Peter Zijlstra
- Bartłomiej Zolnierkiewicz

Poi ci sono altre guide sulla comunità che sono di interesse per molti degli sviluppatori:

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/changes.rst

**Translator** Federico Vaga <[federico.vaga@vaga.pv.it](mailto:federico.vaga@vaga.pv.it)>

## Requisiti minimi per compilare il kernel

### Introduzione

Questo documento fornisce una lista dei software necessari per eseguire i kernel 4.x.

Questo documento è basato sul file “Changes” del kernel 2.0.x e quindi le persone che lo scrissero meritano credito (Jared Mauch, Axel Boldt, Alessandro Sigala, e tanti altri nella rete).

### Requisiti minimi correnti

Prima di pensare d’aver trovato un baco, aggiornate i seguenti programmi **almeno** alla versione indicata! Se non siete certi della versione che state usando, il comando indicato dovrebbe dirvelo.

Questa lista presume che abbiate già un kernel Linux funzionante. In aggiunta, non tutti gli strumenti sono necessari ovunque; ovviamente, se non avete una PC Card, per esempio, probabilmente non dovreste preoccuparvi di pcmciautils.

Programma	Versione minima	Comando per verificare la versione
GNU C	4.9	gcc -version
Clang/LLVM (optional)	10.0.1	clang -version
GNU make	3.81	make -version
binutils	2.23	ld -v
flex	2.5.35	flex -version
bison	2.0	bison -version
util-linux	2.10o	fdformat -version
kmod	13	depmod -V
e2fsprogs	1.41.4	e2fsck -V
jfsutils	1.1.3	fsck.jfs -V
reiserfsprogs	3.6.3	reiserfsck -V
xfsprogs	2.6.0	xfs_db -V
squashfs-tools	4.0	mksquashfs -version
btrfs-progs	0.18	btrfsck
pcmciautils	004	pccardctl -V
quota-tools	3.09	quota -V
PPP	2.4.0	pppd -version
nfs-utils	1.0.5	showmount -version
procps	3.2.0	ps -version
udev	081	udevd -version
grub	0.93	grub -version    grub-install -version
mcelog	0.6	mcelog -version
iptables	1.4.2	iptables -V
openssl & libcrypto	1.0.0	openssl version
bc	1.06.95	bc -version
Sphinx <sup>1</sup>	1.7	sphinx-build -version

<sup>1</sup> Sphinx è necessario solo per produrre la documentazione del Kernel

### Compilazione del kernel

#### GCC

La versione necessaria di gcc potrebbe variare a seconda del tipo di CPU nel vostro calcolatore.

#### Clang/LLVM (opzionale)

L'ultima versione di clang e *LLVM utils* (secondo [releases.llvm.org](http://releases.llvm.org)) sono supportati per la generazione del kernel. Non garantiamo che anche i rilasci più vecchi funzionino, inoltre potremmo rimuovere gli espedienti che abbiamo implementato per farli funzionare. Per maggiori informazioni Building Linux with Clang/LLVM.

#### Make

Per compilare il kernel vi servirà GNU make 3.81 o successivo.

#### Binutils

Per generare il kernel è necessario avere Binutils 2.23 o superiore.

#### pkg-config

Il sistema di compilazione, dalla versione 4.18, richiede pkg-config per verificare l'esistenza degli strumenti kconfig e per determinare le impostazioni da usare in 'make {g,x}config'. Precedentemente pkg-config veniva usato ma non verificato o documentato.

#### Flex

Dalla versione 4.16, il sistema di compilazione, durante l'esecuzione, genera un analizzatore lessicale. Questo richiede flex 2.5.35 o successivo.

#### Bison

Dalla versione 4.16, il sistema di compilazione, durante l'esecuzione, genera un parsificatore. Questo richiede bison 2.0 o successivo.

## Perl

Per compilare il kernel vi servirà perl 5 e i seguenti moduli Getopt::Long, Getopt::Std, File::Basename, e File::Find.

## BC

Vi servirà bc per compilare i kernel dal 3.10 in poi.

## OpenSSL

Il programma OpenSSL e la libreria crypto vengono usati per la firma dei moduli e la gestione dei certificati; sono usati per la creazione della chiave e la generazione della firma.

Se la firma dei moduli è abilitata, allora vi servirà openssl per compilare il kernel 3.7 e successivi. Vi serviranno anche i pacchetti di sviluppo di openssl per compilare il kernel 4.3 o successivi.

## Strumenti di sistema

### Modifiche architetturali

DevFS è stato reso obsoleto da udev (<http://www.kernel.org/pub/linux/utils/kernel/hotplug/>)

Il supporto per UID a 32-bit è ora disponibile. Divertitevi!

La documentazione delle funzioni in Linux è una fase di transizione verso una documentazione integrata nei sorgenti stessi usando dei commenti formattati in modo speciale e posizionati vicino alle funzioni che descrivono. Al fine di arricchire la documentazione, questi commenti possono essere combinati con i file ReST presenti in Documentation/; questi potranno poi essere convertiti in formato PostScript, HTML, LaTex, ePUB o PDF. Per convertire i documenti da ReST al formato che volete, avete bisogno di Sphinx.

## Util-linux

Le versioni più recenti di util-linux: forniscono il supporto a fdisk per dischi di grandi dimensioni; supportano le nuove opzioni di mount; riconoscono più tipi di partizioni; hanno un fdformat che funziona con i kernel 2.4; e altre chicche. Probabilmente vorrete aggiornarlo.

### Ksymoops

Se l'impensabile succede e il kernel va in oops, potrebbe servirvi lo strumento ksymoops per decodificarlo, ma nella maggior parte dei casi non vi servirà. Generalmente è preferibile compilare il kernel con l'opzione `CONFIG_KALLSYMS` cosicché venga prodotto un output più leggibile che può essere usato così com'è (produce anche un output migliore di ksymoops). Se per qualche motivo il vostro kernel non è stato compilato con `CONFIG_KALLSYMS` e non avete modo di ricompilerlo e riprodurre l'oops con quell'opzione abilitata, allora potete usare ksymoops per decodificare l'oops.

### Mkinitrd

I cambiamenti della struttura in `/lib/modules` necessita l'aggiornamento di mkinitrd.

### E2fsprogs

L'ultima versione di `e2fsprogs` corregge diversi bachi in fsck e debugfs. Ovviamente, aggiornarlo è una buona idea.

### JFSutils

Il pacchetto `jfsutils` contiene programmi per il file-system JFS. Sono disponibili i seguenti strumenti:

- `fsck.jfs` - avvia la ripetizione del log delle transizioni, e verifica e ripara una partizione formattata secondo JFS
- `mkfs.jfs` - crea una partizione formattata secondo JFS
- sono disponibili altri strumenti per il file-system.

### Reiserfsprogs

Il pacchetto `reiserfsprogs` dovrebbe essere usato con reiserfs-3.6.x (Linux kernel 2.4.x). Questo è un pacchetto combinato che contiene versioni funzionanti di `mkreiserfs`, `resize_reiserfs`, `debugreiserfs` e `reiserfsck`. Questi programmi funzionano sulle piattaforme i386 e alpha.

### Xfsprogs

L'ultima versione di `xfsprogs` contiene, fra i tanti, i programmi `mkfs.xfs`, `xfs_db` e `xfs_repair` per il file-system XFS. Dipendono dell'architettura e qualsiasi versione dalla 2.0.0 in poi dovrebbe funzionare correttamente con la versione corrente del codice XFS nel kernel (sono raccomandate le versioni 2.6.0 o successive per via di importanti miglioramenti).

## PCMCIAutils

PCMCIAutils sostituisce `pcmica-cs`. Serve ad impostare correttamente i connettori PCMCIA all'avvio del sistema e a caricare i moduli necessari per i dispositivi a 16-bit se il kernel è stato modularizzato e il sottosistema hotplug è in uso.

## Quota-tools

Il supporto per uid e gid a 32 bit richiedono l'uso della versione 2 del formato quota. La versione 3.07 e successive di quota-tools supportano questo formato. Usate la versione raccomandata nella lista qui sopra o una successiva.

## Micro codice per Intel IA32

Per poter aggiornare il micro codice per Intel IA32, è stato aggiunto un apposito driver; il driver è accessibile come un normale dispositivo a caratteri (`misc`). Se non state usando udev probabilmente sarà necessario eseguire i seguenti comandi come root prima di poterlo aggiornare:

```
mkdir /dev/cpu
mknod /dev/cpu/microcode c 10 184
chmod 0644 /dev/cpu/microcode
```

Probabilmente, vorrete anche il programma `microcode_ctl` da usare con questo dispositivo.

## udev

udev è un programma in spazio utente il cui scopo è quello di popolare dinamicamente la cartella `/dev` coi dispositivi effettivamente presenti. udev sostituisce le funzionalità base di `devfs`, consentendo comunque nomi persistenti per i dispositivi.

## FUSE

Serve libfuse 2.4.0 o successiva. Il requisito minimo assoluto è 2.3.0 ma le opzioni di mount `direct_io` e `kernel_cache` non funzioneranno.

## Rete

### Cambiamenti generali

Se per quanto riguarda la configurazione di rete avete esigenze di un certo livello dovreste prendere in considerazione l'uso degli strumenti in `ip-route2`.

### Filtro dei pacchetti / NAT

Il codice per filtraggio dei pacchetti e il NAT fanno uso degli stessi strumenti come nelle versioni del kernel antecedenti la 2.4.x (iptables). Include ancora moduli di compatibilità per 2.2.x ipchains e 2.0.x ipdwadm.

### PPP

Il driver per PPP è stato ristrutturato per supportare collegamenti multipli e per funzionare su diversi livelli. Se usate PPP, aggiornate pppd almeno alla versione 2.4.0.

Se non usate udev, dovete avere un file /dev/ppp che può essere creato da root col seguente comando:

```
mknod /dev/ppp c 108 0
```

### NFS-utils

Nei kernel più antichi (2.4 e precedenti), il server NFS doveva essere informato sui clienti ai quali si voleva fornire accesso via NFS. Questa informazione veniva passata al kernel quando un cliente montava un file-system mediante mountd, oppure usando exportfs all'avvio del sistema. exportfs prende le informazioni circa i clienti attivi da /var/lib/nfs/rmtab.

Questo approccio è piuttosto delicato perché dipende dalla correttezza di rmtab, che non è facile da garantire, in particolare quando si cerca di implementare un *failover*. Anche quando il sistema funziona bene, rmtab ha il problema di accumulare vecchie voci inutilizzate.

Sui kernel più recenti il kernel ha la possibilità di informare mountd quando arriva una richiesta da una macchina sconosciuta, e mountd può dare al kernel le informazioni corrette per l'esportazione. Questo rimuove la dipendenza con rmtab e significa che il kernel deve essere al corrente solo dei clienti attivi.

Per attivare questa funzionalità, dovete eseguire il seguente comando prima di usare exportfs o mountd:

```
mount -t nfsd nfsd /proc/fs/nfsd
```

Dove possibile, raccomandiamo di proteggere tutti i servizi NFS dall'accesso via internet mediante un firewall.

### mcelog

Quando CONFIG\_x86\_MCE è attivo, il programma mcelog processa e registra gli eventi *machine check*. Gli eventi *machine check* sono errori riportati dalla CPU. Incoraggiamo l'analisi di questi errori.

## Documentazione del kernel

### Sphinx

Per i dettagli sui requisiti di Sphinx, fate riferimento a *Installazione Sphinx* in [Documentation/translations/it\\_IT/doc-guide/sphinx.rst](#)

## Ottenere software aggiornato

### Compilazione del kernel

#### gcc

- <<ftp://ftp.gnu.org/gnu/gcc/>>

#### Clang/LLVM

- Getting LLVM.

#### Make

- <<ftp://ftp.gnu.org/gnu/make/>>

#### Binutils

- <<https://www.kernel.org/pub/linux/devel/binutils/>>

#### Flex

- <<https://github.com/westes/flex/releases>>

#### Bison

- <<ftp://ftp.gnu.org/gnu/bison/>>

### OpenSSL

- <<https://www.openssl.org/>>

### Strumenti di sistema

#### Util-linux

- <<https://www.kernel.org/pub/linux/utils/util-linux/>>

#### Kmod

- <<https://www.kernel.org/pub/linux/utils/kernel/kmod/>>
- <<https://git.kernel.org/pub/scm/utils/kernel/kmod/kmod.git>>

#### Ksymoops

- <<https://www.kernel.org/pub/linux/utils/kernel/ksymoops/v2.4/>>

#### Mkinitrd

- <<https://code.launchpad.net/initrd-tools/main>>

#### E2fsprogs

- <<https://www.kernel.org/pub/linux/kernel/people/tytso/e2fsprogs/>>
- <<https://git.kernel.org/pub/scm/fs/ext2/e2fsprogs.git>>

#### JFSutils

- <<http://jfs.sourceforge.net/>>

#### Reiserfsprogs

- <<https://git.kernel.org/pub/scm/linux/kernel/git/jeffm/reiserfsprogs.git>>

## Xfsprogs

- <<https://git.kernel.org/pub/scm/fs/xfs/xfsprogs-dev.git>>
- <<https://www.kernel.org/pub/linux/utils/fs/xfs/xfsprogs/>>

## Pcmciautils

- <<https://www.kernel.org/pub/linux/utils/kernel pcmcia/>>

## Quota-tools

- <<http://sourceforge.net/projects/linuxquota/>>

## Microcodice Intel P6

- <<https://downloadcenter.intel.com/>>

## udev

- <<http://www.freedesktop.org/software/systemd/man/udev.html>>

## FUSE

- <<https://github.com/libfuse/libfuse/releases>>

## mcelog

- <<http://www.mcelog.org/>>

## Rete

## PPP

- <<https://download.samba.org/pub/ppp/>>
- <<https://git.ozlabs.org/?p=ppp.git>>
- <<https://github.com/paulusmack/ppp/>>

### NFS-utils

- <[http://sourceforge.net/project/showfiles.php?group\\_id=14](http://sourceforge.net/project/showfiles.php?group_id=14)>

### Iptables

- <<https://netfilter.org/projects/iptables/index.html>>

### Ip-route2

- <<https://www.kernel.org/pub/linux/utils/net/iproute2/>>

### OProfile

- <<http://oprofile.sf.net/download/>>

### NFS-Utils

- <<http://nfs.sourceforge.net/>>

### Documentazione del kernel

### Sphinx

- <<http://www.sphinx-doc.org/>>

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/submitting-drivers.rst

**Translator** Federico Vaga <[federico.vaga@vaga.pv.it](mailto:federico.vaga@vaga.pv.it)>

### Sottomettere driver per il kernel Linux

---

**Note:** Questo documento è vecchio e negli ultimi anni non è stato più aggiornato; dovrebbe essere aggiornato, o forse meglio, rimosso. La maggior parte di quello che viene detto qui può essere trovato anche negli altri documenti dedicati allo sviluppo. Per questo motivo il documento non verrà tradotto.

---

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/stable-api-nonsense.rst

**Translator** Federico Vaga <[federico.vaga@vaga.pv.it](mailto:federico.vaga@vaga.pv.it)>

## L'interfaccia dei driver per il kernel Linux

(tutte le risposte alle vostre domande e altro)

Greg Kroah-Hartman <[greg@kroah.com](mailto:greg@kroah.com)>

Questo è stato scritto per cercare di spiegare perché Linux **non ha un'interfaccia binaria, e non ha nemmeno un'interfaccia stabile**.

---

**Note:** Questo articolo parla di interfacce **interne al kernel**, non delle interfacce verso lo spazio utente.

L'interfaccia del kernel verso lo spazio utente è quella usata dai programmi, ovvero le chiamate di sistema. Queste interfacce sono **molto** stabili nel tempo e non verranno modificate. Ho vecchi programmi che sono stati compilati su un kernel 0.9 (circa) e tuttora funzionano sulle versioni 2.6 del kernel. Queste interfacce sono quelle che gli utenti e i programmatori possono considerare stabili.

---

## Riepilogo generale

Pensate di volere un'interfaccia del kernel stabile, ma in realtà non la volete, e nemmeno sapete di non volerla. Quello che volete è un driver stabile che funzioni, e questo può essere ottenuto solo se il driver si trova nei sorgenti del kernel. Ci sono altri vantaggi nell'avere il proprio driver nei sorgenti del kernel, ognuno dei quali hanno reso Linux un sistema operativo robusto, stabile e maturo; questi sono anche i motivi per cui avete scelto Linux.

## Introduzione

Solo le persone un po' strambe vorrebbero scrivere driver per il kernel con la costante preoccupazione per i cambiamenti alle interfacce interne. Per il resto del mondo, queste interfacce sono invisibili o non di particolare interesse.

Innanzitutto, non tratterò **alcun** problema legale riguardante codice chiuso, nascosto, avvolto, blocchi binari, o qualsiasi altra cosa che descrive driver che non hanno i propri sorgenti rilasciati con licenza GPL. Per favore fate riferimento ad un avvocato per qualsiasi questione legale, io sono un programmatore e perciò qui vi parlerò soltanto delle questioni tecniche (non per essere superficiali sui problemi legali, sono veri e dovete esserne a conoscenza in ogni circostanza).

Dunque, ci sono due tematiche principali: interfacce binarie del kernel e interfacce stabili nei sorgenti. Ognuna dipende dall'altra, ma discuteremo prima delle cose binarie per toglierle di mezzo.

### Interfaccia binaria del kernel

Supponiamo d'averne un'interfaccia stabile nei sorgenti del kernel, di conseguenza un'interfaccia binaria dovrebbe essere anche essa stabile, giusto? Sbagliato. Prendete in considerazione i seguenti fatti che riguardano il kernel Linux:

- A seconda della versione del compilatore C che state utilizzando, diverse strutture dati del kernel avranno un allineamento diverso, e possibilmente un modo diverso di includere le funzioni (renderle inline oppure no). L'organizzazione delle singole funzioni non è poi così importante, ma la spaziatura (*padding*) nelle strutture dati, invece, lo è.
- In base alle opzioni che sono state selezionate per generare il kernel, un certo numero di cose potrebbero succedere:
  - strutture dati differenti potrebbero contenere campi differenti
  - alcune funzioni potrebbero non essere implementate (per esempio, alcuni *lock* spariscono se compilati su sistemi mono-processore)
  - la memoria interna del kernel può essere allineata in differenti modi a seconda delle opzioni di compilazione.
- Linux funziona su una vasta gamma di architetture di processore. Non esiste alcuna possibilità che il binario di un driver per un'architettura funzioni correttamente su un'altra.

Alcuni di questi problemi possono essere risolti compilando il proprio modulo con la stessa identica configurazione del kernel, ed usando la stessa versione del compilatore usato per compilare il kernel. Questo è sufficiente se volete fornire un modulo per uno specifico rilascio su una specifica distribuzione Linux. Ma moltiplicate questa singola compilazione per il numero di distribuzioni Linux e il numero dei rilasci supportati da quest'ultime e vi troverete rapidamente in un incubo fatto di configurazioni e piattaforme hardware (differenti processori con differenti opzioni); dunque, anche per il singolo rilascio di un modulo, dovreste creare differenti versioni dello stesso.

Fidatevi, se tenterete questa via, col tempo, diventerete pazzi; l'ho imparato a mie spese molto tempo fa...

### Interfaccia stabile nei sorgenti del kernel

Se parlate con le persone che cercano di mantenere aggiornato un driver per Linux ma che non si trova nei sorgenti, allora per queste persone l'argomento sarà "ostico".

Lo sviluppo del kernel Linux è continuo e viaggia ad un ritmo sostenuto, e non rallenta mai. Perciò, gli sviluppatori del kernel trovano bachi nelle interfacce attuali, o trovano modi migliori per fare le cose. Se le trovano, allora le correggeranno per migliorarle. In questo frangente, i nomi delle funzioni potrebbero cambiare, le strutture dati potrebbero diventare più grandi o più piccole, e gli argomenti delle funzioni potrebbero essere ripensati. Se questo dovesse succedere, nello stesso momento, tutte le istanze dove questa interfaccia viene utilizzata verranno corrette, garantendo che tutto continui a funzionare senza problemi.

Portiamo ad esempio l'interfaccia interna per il sottosistema USB che ha subito tre ristrutturazioni nel corso della sua vita. Queste ristrutturazioni furono fatte per risolvere diversi problemi:

- È stato fatto un cambiamento da un flusso di dati sincrono ad uno asincrono. Questo ha ridotto la complessità di molti driver e ha aumentato la capacità di trasmissione di tutti i driver fino a raggiungere quasi la velocità massima possibile.
- È stato fatto un cambiamento nell'allocazione dei pacchetti da parte del sottosistema USB per conto dei driver, cosicché ora i driver devono fornire più informazioni al sottosistema USB al fine di correggere un certo numero di stalli.

Questo è completamente l'opposto di quello che succede in alcuni sistemi operativi proprietari che hanno dovuto mantenere, nel tempo, il supporto alle vecchie interfacce USB. I nuovi sviluppatori potrebbero usare accidentalmente le vecchie interfacce e sviluppare codice nel modo sbagliato, portando, di conseguenza, all'instabilità del sistema.

In entrambe gli scenari, gli sviluppatori hanno ritenuto che queste importanti modifiche erano necessarie, e quindi le hanno fatte con qualche sofferenza. Se Linux avesse assicurato di mantenere stabile l'interfaccia interna, si sarebbe dovuto procedere alla creazione di una nuova, e quelle vecchie, e mal funzionanti, avrebbero dovuto ricevere manutenzione, creando lavoro aggiuntivo per gli sviluppatori del sottosistema USB. Dato che gli sviluppatori devono dedicare il proprio tempo a questo genere di lavoro, chiedergli di dedicarne dell'altro, senza benefici, magari gratuitamente, non è contemplabile.

Le problematiche relative alla sicurezza sono molto importanti per Linux. Quando viene trovato un problema di sicurezza viene corretto in breve tempo. A volte, per prevenire il problema di sicurezza, si sono dovute cambiare delle interfacce interne al kernel. Quando è successo, allo stesso tempo, tutti i driver che usavano quelle interfacce sono stati aggiornati, garantendo la correzione definitiva del problema senza doversi preoccupare di rivederlo per sbaglio in futuro. Se non si fossero cambiate le interfacce interne, sarebbe stato impossibile correggere il problema e garantire che non si sarebbe più ripetuto.

Nel tempo le interfacce del kernel subiscono qualche ripulita. Se nessuno sta più usando un'interfaccia, allora questa verrà rimossa. Questo permette al kernel di rimanere il più piccolo possibile, e garantisce che tutte le potenziali interfacce sono state verificate nel limite del possibile (le interfacce inutilizzate sono impossibili da verificare).

## Cosa fare

Dunque, se avete un driver per il kernel Linux che non si trova nei sorgenti principali del kernel, come sviluppatori, cosa dovreste fare? Rilasciare un file binario del driver per ogni versione del kernel e per ogni distribuzione, è un incubo; inoltre, tenere il passo con tutti i cambiamenti del kernel è un brutto lavoro.

Semplicemente, fate sì che il vostro driver per il kernel venga incluso nei sorgenti principali (ricordatevi, stiamo parlando di driver rilasciati secondo una licenza compatibile con la GPL; se il vostro codice non ricade in questa categoria: buona fortuna, arrangiatevi, siete delle sanguisughe)

Se il vostro driver è nei sorgenti del kernel e un'interfaccia cambia, il driver verrà corretto immediatamente dalla persona che l'ha modificata. Questo garantisce che sia sempre possibile compilare il driver, che funzioni, e tutto con un minimo sforzo da parte vostra.

Avere il proprio driver nei sorgenti principali del kernel ha i seguenti vantaggi:

- La qualità del driver aumenterà e i costi di manutenzione (per lo sviluppatore originale) diminuiranno.

- Altri sviluppatori aggiungeranno nuove funzionalità al vostro driver.
- Altri persone troveranno e correggeranno bachi nel vostro driver.
- Altri persone troveranno degli aggiustamenti da fare al vostro driver.
- Altri persone aggiorneranno il driver quando è richiesto da un cambiamento di un'interfaccia.
- Il driver sarà automaticamente reso disponibile in tutte le distribuzioni Linux senza dover chiedere a nessuna di queste di aggiungerlo.

Dato che Linux supporta più dispositivi di qualsiasi altro sistema operativo, e che girano su molti più tipi di processori di qualsiasi altro sistema operativo; ciò dimostra che questo modello di sviluppo qualcosa di giusto, dopo tutto, lo fa :)

---

Dei ringraziamenti vanno a Randy Dunlap, Andrew Morton, David Brownell, Hanna Linder, Robert Love, e Nishanth Aravamudan per la loro revisione e per i loro commenti sulle prime bozze di questo articolo.

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** ../../process/management-style

**Translator** Alessia Mantegazza <[amantergazza@vaga.pv.it](mailto:amantergazza@vaga.pv.it)>

## Il modello di gestione del kernel Linux

Questo breve documento descrive il modello di gestione del kernel Linux. Per certi versi, esso rispecchia il documento [translations/it\\_IT/process/coding-style.rst](#), ed è principalmente scritto per evitare di rispondere<sup>1</sup> in continuazione alle stesse identiche (o quasi) domande.

Il modello di gestione è qualcosa di molto personale e molto più difficile da qualificare rispetto a delle semplici regole di codifica, quindi questo documento potrebbe avere più o meno a che fare con la realtà. È cominciato come un gioco, ma ciò non significa che non possa essere vero. Lo dovrete decidere voi stessi.

In ogni caso, quando si parla del “dirigente del kernel”, ci si riferisce sempre alla persona che dirige tecnicamente, e non a coloro che tradizionalmente hanno un ruolo direttivo all'interno delle aziende. Se vi occupate di convalidare acquisti o avete una qualche idea sul budget del vostro gruppo, probabilmente non siete un dirigente del kernel. Quindi i suggerimenti qui indicati potrebbero fare al caso vostro, oppure no.

Prima di tutto, suggerirei di acquistare “Le sette regole per avere successo”, e di non leggerlo. Bruciatelo, è un grande gesto simbolico.

Comunque, partiamo:

---

<sup>1</sup> Questo documento non fa molto per risponde alla domanda, ma rende così dannatamente ovvio a chi la pone che non abbiamo la minima idea di come rispondere.

## 1) Le decisioni

Tutti pensano che i dirigenti decidano, e che questo prendere decisioni sia importante. Più grande e dolorosa è la decisione, più importante deve essere il dirigente che la prende. Questo è molto profondo ed ovvio, ma non è del tutto vero.

Il gioco consiste nell'“evitare” di dover prendere decisioni. In particolare se qualcuno vi chiede di “Decidere” tra (a) o (b), e vi dice che ha davvero bisogno di voi per questo, come dirigenti siete nei guai. Le persone che gestite devono conoscere i dettagli più di quanto li conosciate voi, quindi se vengono da voi per una decisione tecnica, siete fottuti. Non sarete chiaramente competente per prendere quella decisione per loro.

(Corollario: se le persone che gestite non conoscono i dettagli meglio di voi, anche in questo caso sarete fregati, tuttavia per altre ragioni. Ossia state facendo il lavoro sbagliato, e che invece dovrebbero essere “loro” a gestirvi)

Quindi il gioco si chiama “evitare” decisioni, almeno le più grandi e difficili. Prendere decisioni piccoli e senza conseguenze va bene, e vi fa sembrare competenti in quello che state facendo, quindi quello che un dirigente del kernel ha bisogno di fare è trasformare le decisioni grandi e difficili in minuzie delle quali nessuno importa.

Ciò aiuta a capire che la differenza chiave tra una grande decisione ed una piccola sta nella possibilità di modificare tale decisione in seguito. Qualsiasi decisione importante può essere ridotta in decisioni meno importanti, ma dovete assicurarvi che possano essere reversibili in caso di errori (presenti o futuri). Improvvisamente, dovrete essere doppiamente dirigenti per **due** decisioni non sequenziali - quella sbagliata **e** quella giusta.

E le persone vedranno tutto ciò come prova di vera capacità di comando (*cough cavolata cough*)

Così la chiave per evitare le decisioni difficili diviene l’evitare di fare cose che non possono essere disfatte. Non infilatevi in un angolo dal quale non potrete sfuggire. Un topo messo all’angolo può rivelarsi pericoloso - un dirigente messo all’angolo è solo pietoso.

**In ogni caso** dato che nessuno è stupido al punto da lasciare veramente ad un dirigente del kernel un’enorme responsabilità, solitamente è facile fare marcia indietro. Annullare una decisione è molto facile: semplicemente dite a tutti che siete stati degli scemi incompetenti, dite che siete dispiaciuti, ed annullate tutto l’inutile lavoro sul quale gli altri hanno lavorato nell’ultimo anno. Improvvisamente la decisione che avevate preso un anno fa non era poi così grossa, dato che può essere facilmente annullata.

È emerso che alcune persone hanno dei problemi con questo tipo di approccio, questo per due ragioni:

- ammettere di essere degli idioti è più difficile di quanto sembri. A tutti noi piace mantenere le apparenze, ed uscire allo scoperto in pubblico per ammettere che ci si è sbagliati è qualcosa di davvero impegnativo.
- avere qualcuno che ti dice che ciò su cui hai lavorato nell’ultimo anno non era del tutto valido, può rivelarsi difficile anche per un povero ed umile ingegnere, e mentre il **lavoro** vero era abbastanza facile da cancellare, dall’altro canto potreste aver irrimediabilmente perso la fiducia di quell’ingegnere. E ricordate che l’“irrevocabile” era quello che avevamo cercato di evitare fin dall’inizio, e la vostra decisione ha finito per esserlo.

Fortunatamente, entrambe queste ragioni posso essere mitigate semplicemente ammettendo fin dal principio che non avete una cavolo di idea, dicendo agli altri in anticipo che la vostra decisione è puramente ipotetica, e che potrebbe essere sbagliata. Dovreste sempre riservarvi

il diritto di cambiare la vostra opinione, e rendere gli altri ben **consapevoli** di ciò. Ed è molto più facile ammettere di essere stupidi quando non avete **ancora** fatto quella cosa stupida.

Poi, quando è realmente emersa la vostra stupidità, le persone semplicemente roteeranno gli occhi e diranno “Uffa, no, ancora”.

Questa ammissione preventiva di incompetenza potrebbe anche portare le persone che stanno facendo il vero lavoro, a pensarci due volte. Dopo tutto, se **loro** non sono certi se sia una buona idea, voi, sicuro come la morte, non dovreste incoraggiarli promettendogli che ciò su cui stanno lavorando verrà incluso. Fate sì che ci pensino due volte prima che si imbarchino in un grosso lavoro.

Ricordate: loro devono sapere più cose sui dettagli rispetto a voi, e solitamente pensano di avere già la risposta a tutto. La miglior cosa che potete fare in qualità di dirigente è di non instillare troppa fiducia, ma invece fornire una salutare dose di pensiero critico su quanto stanno facendo.

Comunque, un altro modo di evitare una decisione è quello di lamentarsi malinconicamente dicendo : “non possiamo farli entrambi e basta?” e con uno sguardo pietoso. Fidatevi, funziona. Se non è chiaro quale sia il miglior approccio, lo scopriranno. La risposta potrebbe essere data dal fatto che entrambe i gruppi di lavoro diventano frustati al punto di rinunciarvi.

Questo può suonare come un fallimento, ma di solito questo è un segno che c'era qualcosa che non andava in entrambe i progetti, e il motivo per il quale le persone coinvolte non abbiano potuto decidere era che entrambe sbagliavano. Voi ne uscirete freschi come una rosa, e avrete evitato un'altra decisione con la quale avreste potuto fregarvi.

## 2) Le persone

Ci sono molte persone stupide, ed essere un dirigente significa che dovrete scendere a patti con questo, e molto più importate, che **loro** devono avere a che fare con **voi**.

Ne emerge che mentre è facile annullare degli errori tecnici, non è invece così facile rimuovere i disordini della personalità. Dovrete semplicemente convivere con i loro, ed i vostri, problemi.

Comunque, al fine di preparavi in qualità di dirigenti del kernel, è meglio ricordare di non abbattere alcun ponte, bombardare alcun paesano innocente, o escludere troppi sviluppatori kernel. Ne emerge che escludere le persone è piuttosto facile, mentre includerle nuovamente è difficile. Così “l'esclusione” immediatamente cade sotto il titolo di “non reversibile”, e diviene un no-no secondo la sezione [1\) Le decisioni](#).

Esistono alcune semplici regole qui:

- (1) non chiamate le persone teste di c\*\*\* (al meno, non in pubblico)
- (2) imparate a scusarvi quando dimenticate la regola (1)

Il problema del punto numero 1 è che è molto facile da rispettare, dato che è possibile dire “sei una testa di c\*\*\*” in milioni di modi differenti<sup>2</sup>, a volte senza nemmeno pensarci, e praticamente sempre con la calda convinzione di essere nel giusto.

E più convinti sarete che avete ragione (e diciamolo, potete chiamare praticamente **tutti** testa di c\*\*, e spesso **sarete** nel giusto), più difficile sarà scusarvi successivamente.

Per risolvere questo problema, avete due possibilità:

<sup>2</sup> Paul Simon cantava: “50 modi per lasciare il vostro amante”, perché, molto francamente, “Un milione di modi per dire ad uno sviluppatore Testa di c\*\*\*” non avrebbe funzionato. Ma sono sicuro che ci abbia pensato.

- diventare davvero bravi nello scusarsi
- essere amabili così che nessuno finirà col sentirsi preso di mira. Siate creativi abbastanza, e potrebbero esserne divertiti.

L'opzione dell'essere immancabilmente educati non esiste proprio. Nessuno si fiderà di qualcuno che chiaramente sta nascondendo il suo vero carattere.

### 3) Le persone II - quelle buone

Mentre emerge che la maggior parte delle persone sono stupide, il corollario a questo è il triste fatto che anche voi siete fra queste, e che mentre possiamo tutti crogiolarci nella sicurezza di essere migliori della media delle persone (diciamocelo, nessuno crede di essere nelle media o sotto di essa), dovremmo anche ammettere che non siamo il "coltello più affilato" del circondario, e che ci saranno altre persone che sono meno stupide di quanto lo siete voi.

Molti reagiscono male davanti alle persone intelligenti. Altri le usano a proprio vantaggio.

Assicuratevi che voi, in qualità di manutentori del kernel, siate nel secondo gruppo. Inchinatevi dinanzi a loro perché saranno le persone che vi renderanno il lavoro più facile. In particolare, prenderanno le decisioni per voi, che è l'oggetto di questo gioco.

Quindi quando trovate qualcuno più sveglio di voi, prendetevela comoda. Le vostre responsabilità dirigenziali si ridurranno in gran parte nel dire "Sembra una buona idea - Vai", oppure "Sembra buono, ma invece circa questo e quello?". La seconda versione in particolare è una gran modo per imparare qualcosa di nuovo circa "questo e quello" o di sembrare **extra** dirigenziali sottolineando qualcosa alla quale i più svegli non avevano pensato. In entrambe i casi, vincete.

Una cosa alla quale dovete fare attenzione è che l'essere grandi in qualcosa non si traduce automaticamente nell'essere grandi anche in altre cose. Quindi dovreste dare una spintarella alle persone in una specifica direzione, ma diciamocelo, potrebbero essere bravi in ciò che fanno e far schifo in tutto il resto. La buona notizia è che le persone tendono a gravitare attorno a ciò in cui sono bravi, quindi non state facendo nulla di irreversibile quando li spingete verso una certa direzione, solo non spingete troppo.

### 4) Addossare le colpe

Le cose andranno male, e le persone vogliono qualcuno da incolpare. Sarete voi.

Non è poi così difficile accettare la colpa, specialmente se le persone riescono a capire che non era **tutta** colpa vostra. Il che ci porta sulla miglior strada per assumersi la colpa: fatelo per qualcun'altro. Vi sentirete bene nel assumervi la responsabilità, e loro si sentiranno bene nel non essere incolpati, e coloro che hanno perso i loro 36GB di pornografia a causa della vostra incompetenza ammetteranno a malincuore che almeno non avete cercato di fare il furbetto.

Successivamente fate in modo che gli sviluppatori che in realtà hanno fallito (se riuscite a trovarli) sappiano **in privato** che sono "fottuti". Questo non per fargli sapere che la prossima volta possono evitarselo ma per fargli capire che sono in debito. E, forse cosa più importante, sono loro che devono sistemare la cosa. Perché, ammettiamolo, è sicuro non sarete voi a farlo.

Assumersi la colpa è anche ciò che vi rendere dirigenti in prima battuta. È parte di ciò che spinge gli altri a fidarsi di voi, e vi garantisce la gloria potenziale, perché siete gli unici a dire

“Ho fatto una cavolata”. E se avete seguito le regole precedenti, sarete decisamente bravi nel dirlo.

### 5) Le cose da evitare

Esiste una cosa che le persone odiano più che essere chiamate “teste di c\*\*\*\*”, ed è essere chiamate “teste di c\*\*\*\*” con fare da bigotto. Se per il primo caso potrete comunque scusarvi, per il secondo non ve ne verrà data nemmeno l’opportunità. Probabilmente smetteranno di ascoltarvi anche se tutto sommato state svolgendo un buon lavoro.

Tutti crediamo di essere migliori degli altri, il che significa che quando qualcuno inizia a darsi delle arie, ci da **davvero** fastidio. Potreste anche essere moralmente ed intellettualmente superiore a tutti quelli attorno a voi, ma non cercate di renderlo ovvio per gli altri a meno che non **vogliate** veramente far arrabbiare qualcuno<sup>3</sup>.

Allo stesso modo evitate di essere troppo gentili e pacati. Le buone maniere facilmente finiscono per strabordare e nascondere i problemi, e come si usa dire, “su internet nessuno può sentire la vostra pacatezza”. Usate argomenti diretti per farvi capire, non potete sperare che la gente capisca in altro modo.

Un po’ di umorismo può aiutare a smorzare sia la franchezza che la moralità. Andare oltre i limiti al punto d’essere ridicolo può portare dei punti a casa senza renderlo spiacevole per i riceventi, i quali penseranno che stavate facendo gli scemi. Può anche aiutare a lasciare andare quei blocchi mentali che abbiamo nei confronti delle critiche.

### 6) Perché io?

Dato che la vostra responsabilità principale è quella di prendervi le colpe d’altri, e rendere dolorosamente ovvio a tutti che siete degli incompetenti, la domanda naturale che ne segue sarà : perché dovrei fare tutto ciò?

Innanzitutto, potreste diventare o no popolari al punto da avere la fila di ragazzine (o ragazzini, evitiamo pregiudizi o sessismo) che gridano e bussano alla porta del vostro camerino, ma comunque **proverete** un immenso senso di realizzazione personale dall’essere “in carica”. Dimenticate il fatto che voi state discutendo con tutti e che cercate di inseguirli il più velocemente che potete. Tutti continueranno a pensare che voi siete la persona in carica.

È un bel lavoro se riuscite ad adattarlo a voi.

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l’unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/stable-kernel-rules.rst

**Translator** Federico Vaga <[federico.vaga@vaga.pv.it](mailto:federico.vaga@vaga.pv.it)>

<sup>3</sup> Suggerimento: i forum di discussione su internet, che non sono collegati col vostro lavoro, sono ottimi modi per sfogare la frustrazione verso altre persone. Di tanto in tanto scrivete messaggi offensivi col ghigno in faccia per infiammare qualche discussione: vi sentirete purificati. Solo cercate di non cagare troppo vicino a casa.

## Tutto quello che volevate sapere sui rilasci -stable di Linux

Regole sul tipo di patch che vengono o non vengono accettate nei sorgenti “-stable”:

- Ovviamente dev'essere corretta e verificata.
- Non dev'essere più grande di 100 righe, incluso il contesto.
- Deve correggere una cosa sola.
- Deve correggere un baco vero che sta disturbando gli utenti (non cose del tipo “Questo potrebbe essere un problema ...”).
- Deve correggere un problema di compilazione (ma non per cose già segnate con CONFIG\_BROKEN), un kernel oops, un blocco, una corruzione di dati, un vero problema di sicurezza, o problemi del tipo “oh, questo non va bene”. In pratica, qualcosa di critico.
- Problemi importanti riportati dagli utenti di una distribuzione potrebbero essere considerati se correggono importanti problemi di prestazioni o di interattività. Dato che questi problemi non sono così ovvi e la loro correzione ha un'alta probabilità d'introdurre una regressione, dovrebbero essere sottomessi solo dal manutentore della distribuzione includendo un link, se esiste, ad un rapporto su bugzilla, e informazioni aggiuntive sull'impatto che ha sugli utenti.
- Non deve correggere problemi relativi a una “teorica sezione critica”, a meno che non venga fornita anche una spiegazione su come questa si possa verificare.
- Non deve includere alcuna correzione “banale” (correzioni grammaticali, pulizia dagli spazi bianchi, eccetera).
- Deve rispettare le regole scritte in [Documentation/translations/it\\_IT/process/submitting-patches.rst](#)
- Questa patch o una equivalente deve esistere già nei sorgenti principali di Linux

## Procedura per sottomettere patch per i sorgenti -stable

- Una patch di sicurezza non dovrebbero essere gestite (solamente) dal processo di revisione -stable, ma dovrebbe seguire le procedure descritte in [Documentation/translations/it\\_IT/admin-guide/security-bugs.rst](#).

## Per tutte le altre sottomissioni, scegliere una delle seguenti procedure

### Opzione 1

Per far sì che una patch venga automaticamente inclusa nei sorgenti stabili, aggiungete l'etichetta

Cc: stable@vger.kernel.org

nell'area dedicata alla firme. Una volta che la patch è stata inclusa, verrà applicata anche sui sorgenti stabili senza che l'autore o il manutentore del sottosistema debba fare qualcosa.

### Opzione 2

Dopo che la patch è stata inclusa nei sorgenti Linux, inviate una mail a [stable@vger.kernel.org](mailto:stable@vger.kernel.org) includendo: il titolo della patch, l'identificativo del commit, il perché pensate che debba essere applicata, e in quale versione del kernel la vorreste vedere.

### Opzione 3

Inviata la patch, dopo aver verificato che rispetta le regole descritte in precedenza, a [stable@vger.kernel.org](mailto:stable@vger.kernel.org). Dovete annotare nel changelog l'identificativo del commit nei sorgenti principali, così come la versione del kernel nel quale vorreste vedere la patch.

L'[Opzione 1](#) è fortemente raccomandata; è il modo più facile e usato. L'[Opzione 2](#) e l'[Opzione 3](#) sono più utili quando, al momento dell'inclusione dei sorgenti principali, si ritiene che non debbano essere incluse anche in quelli stabili (per esempio, perché si crede che si dovrebbero fare più verifiche per eventuali regressioni). L'[Opzione 3](#) è particolarmente utile se la patch ha bisogno di qualche modifica per essere applicata ad un kernel più vecchio (per esempio, perché nel frattempo l'API è cambiata).

Notate che per l'[Opzione 3](#), se la patch è diversa da quella nei sorgenti principali (per esempio perché è stato necessario un lavoro di adattamento) allora dev'essere ben documentata e giustificata nella descrizione della patch.

L'identificativo del commit nei sorgenti principali dev'essere indicato sopra al messaggio della patch, così:

```
commit <sha1> upstream.
```

In aggiunta, alcune patch inviate attraverso l'[Opzione 1](#) potrebbero dipendere da altre che devo essere incluse. Questa situazione può essere indicata nel seguente modo nell'area dedicata alle firme:

```
Cc: <stable@vger.kernel.org> # 3.3.x: a1f84a3: sched: Check for idle
Cc: <stable@vger.kernel.org> # 3.3.x: 1b9508f: sched: Rate-limit newidle
Cc: <stable@vger.kernel.org> # 3.3.x: fd21073: sched: Fix affinity logic
Cc: <stable@vger.kernel.org> # 3.3.x
Signed-off-by: Ingo Molnar <mingo@elte.hu>
```

La sequenza di etichette ha il seguente significato:

```
git cherry-pick a1f84a3
git cherry-pick 1b9508f
git cherry-pick fd21073
git cherry-pick <this commit>
```

Inoltre, alcune patch potrebbero avere dei requisiti circa la versione del kernel. Questo può essere indicato usando il seguente formato nell'area dedicata alle firme:

```
Cc: <stable@vger.kernel.org> # 3.3.x
```

L'etichetta ha il seguente significato:

```
git cherry-pick <this commit>
```

per ogni sorgente “-stable” che inizia con la versione indicata.

Dopo la sottomissione:

- Il mittente riceverà un ACK quando la patch è stata accettata e messa in coda, oppure un NAK se la patch è stata rigettata. A seconda degli impegni degli sviluppatori, questa risposta potrebbe richiedere alcuni giorni.
- Se accettata, la patch verrà aggiunta alla coda -stable per essere revisionata dai altri sviluppatori e dal principale manutentore del sottosistema.

## Ciclo di una revisione

- Quando i manutentori -stable decidono di fare un ciclo di revisione, le patch vengono mandate al comitato per la revisione, ai manutentori soggetti alle modifiche delle patch (a meno che il mittente non sia anche il manutentore di quell’area del kernel) e in CC: alla lista di discussione linux-kernel.
- La commissione per la revisione ha 48 ore per dare il proprio ACK o NACK alle patch.
- Se una patch viene rigettata da un membro della commissione, o un membro della lista linux-kernel obietta la bontà della patch, sollevando problemi che i manutentori ed i membri non avevano compreso, allora la patch verrà rimossa dalla coda.
- Alla fine del ciclo di revisione tutte le patch che hanno ricevuto l’ACK verranno aggiunte per il prossimo rilascio -stable, e successivamente questo nuovo rilascio verrà fatto.
- Le patch di sicurezza verranno accettate nei sorgenti -stable direttamente dalla squadra per la sicurezza del kernel, e non passerà per il normale ciclo di revisione. Contattate la suddetta squadra per maggiori dettagli su questa procedura.

## Sorgenti

- La coda delle patch, sia quelle già applicate che in fase di revisione, possono essere trovate al seguente indirizzo:

<https://git.kernel.org/pub/scm/linux/kernel/git/stable/stable-queue.git>

- Il rilascio definitivo, e marchiato, di tutti i kernel stabili può essere trovato in rami distinti per versione al seguente indirizzo:

<https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git>

### Comitato per la revisione

- Questo comitato è fatto di sviluppatori del kernel che si sono offerti volontari per questo lavoro, e pochi altri che non sono proprio volontari.

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/submit-checklist.rst

**Translator** Federico Vaga <[federico.vaga@vaga.pv.it](mailto:federico.vaga@vaga.pv.it)>

### Lista delle verifiche da fare prima di inviare una patch per il kernel Linux

Qui troverete una lista di cose che uno sviluppatore dovrebbe fare per vedere le proprie patch accettate più rapidamente.

Tutti questi punti integrano la documentazione fornita riguardo alla sottomissione delle patch, in particolare [Documentation/translations/it\\_IT/process/submitting-patches.rst](#).

- 1) Se state usando delle funzionalità del kernel allora includete (#include) i file che le dichiarano/definiscono. Non dipendente dal fatto che un file d'intestazione include anche quelli usati da voi.
- 2) Compilazione pulita:
  - a) con le opzioni CONFIG negli stati =y, =m e =n. Nessun avviso/errore di gcc e nessun avviso/errore dal linker.
  - b) con `allnoconfig`, `allmodconfig`
  - c) quando si usa `0=builddir`
  - d) Qualsiasi modifica in Documentation/ deve compilare con successo senza avvisi o errori. Usare `make htmldocs` o `make pdfdocs` per verificare e correggere i problemi
- 3) Compilare per diverse architetture di processore usando strumenti per la cross-compilazione o altri.
- 4) Una buona architettura per la verifica della cross-compilazione è la ppc64 perché tende ad usare `unsigned long` per le quantità a 64-bit.
- 5) Controllate lo stile del codice della vostra patch secondo le direttive scritte in [Documentation/translations/it\\_IT/process/coding-style.rst](#). Prima dell'invio della patch, usate il verificatore di stile (`script/checkpatch.pl`) per scovare le violazioni più semplici. Dovreste essere in grado di giustificare tutte le violazioni rimanenti nella vostra patch.
- 6) Le opzioni CONFIG, nuove o modificate, non scompongono il menu di configurazione e sono preimpostate come disabilitate a meno che non soddisfino i criteri descritti in [Documentation/kbuild/kconfig-language.rst](#) alla punto "Voci di menu: valori pre-definiti".
- 7) Tutte le nuove opzioni Kconfig hanno un messaggio di aiuto.

- 8) La patch è stata accuratamente revisionata rispetto alle più importanti configurazioni Kconfig. Questo è molto difficile da fare correttamente - un buono lavoro di testa sarà utile.
- 9) Verificare con sparse.
- 10) Usare `make checkstack` e correggere tutti i problemi rilevati.

---

**Note:** `checkstack` non evidenzia esplicitamente i problemi, ma una funzione che usa più di 512 byte sullo stack è una buona candidata per una correzione.

---

- 11) Includete commenti kernel-doc per documentare API globali del kernel. Usate `make htmldocs` o `make pdfdocs` per verificare i commenti kernel-doc ed eventualmente correggerli.
- 12) La patch è stata verificata con le seguenti opzioni abilitate contemporaneamente: `CONFIG_PREEMPT`, `CONFIG_DEBUG_PREEMPT`, `CONFIG_DEBUG_SLAB`, `CONFIG_DEBUG_PAGEALLOC`, `CONFIG_DEBUG_MUTEXES`, `CONFIG_DEBUG_SPINLOCK`, `CONFIG_DEBUG_ATOMIC_SLEEP`, `CONFIG_PROVE_RCU` e `CONFIG_DEBUG_OBJECTS_RCU_HEAD`.
- 13) La patch è stata compilata e verificata in esecuzione con, e senza, le opzioni `CONFIG_SMP` e `CONFIG_PREEMPT`.
- 14) Se la patch ha effetti sull'IO dei dischi, eccetera: allora dev'essere verificata con, e senza, l'opzione `CONFIG_LBDAF`.
- 15) Tutti i percorsi del codice sono stati verificati con tutte le funzionalità di lockdep abilitate.
- 16) Tutti i nuovi elementi in `/proc` sono documentati in `Documentation/`.
- 17) Tutti i nuovi parametri d'avvio del kernel sono documentati in `Documentation/admin-guide/kernel-parameters.rst`.
- 18) Tutti i nuovi parametri dei moduli sono documentati con `MODULE_PARM_DESC()`.
- 19) Tutte le nuove interfacce verso lo spazio utente sono documentate in `Documentation/ABI/`. Leggete `Documentation/ABI/README` per maggiori informazioni. Le patch che modificano le interfacce utente dovrebbero essere inviate in copia anche a [linux-api@vger.kernel.org](mailto:linux-api@vger.kernel.org).
- 20) La patch è stata verificata con l'iniezione di fallimenti in slab e nell'allocazione di pagine. Vedere `Documentation/fault-injection/`.  
Se il nuovo codice è corposo, potrebbe essere opportuno aggiungere l'iniezione di fallimenti specifici per il sottosistema.
- 21) Il nuovo codice è stato compilato con `gcc -W` (usate `make KCFLAGS=-W`). Questo genererà molti avvisi, ma è ottimo per scovare bachi come "warning: comparison between signed and unsigned".
- 22) La patch è stata verificata dopo essere stata inclusa nella serie di patch -mm; questo al fine di assicurarsi che continui a funzionare assieme a tutte le altre patch in coda e i vari cambiamenti nei sottosistemi VM, VFS e altri.
- 23) Tutte le barriere di sincronizzazione {per esempio, `barrier()`, `rmb()`, `wmb()`} devono essere accompagnate da un commento nei sorgenti che ne spieghi la logica: cosa fanno e perché.

- 24) Se la patch aggiunge nuove chiamate ioctl, allora aggiornate Documentation/userspace-api/ioctl/ioctl-number.rst.
- 25) Se il codice che avete modificato dipende o usa una qualsiasi interfaccia o funzionalità del kernel che è associata a uno dei seguenti simboli Kconfig, allora verificate che il kernel compili con diverse configurazioni dove i simboli sono disabilitati e/o =m (se c'è la possibilità) [non tutti contemporaneamente, solo diverse combinazioni casuali]:  
CONFIG\_SMP, CONFIG\_SYSFS, CONFIG\_PROC\_FS, CONFIG\_INPUT, CONFIG\_PCI, CONFIG\_BLOCK, CONFIG\_PM, CONFIG\_MAGIC\_SYSRQ, CONFIG\_NET, CONFIG\_INET=n (ma l'ultimo con CONFIG\_NET=y).

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/kernel-docs.rst

**Translator** Federico Vaga <[federico.vaga@vaga.pv.it](mailto:federico.vaga@vaga.pv.it)>

## Indice di documenti per le persone interessate a capire e/o scrivere per il kernel Linux

---

**Note:** Questo documento contiene riferimenti a documenti in lingua inglese; inoltre utilizza dai campi *ReStructuredText* di supporto alla ricerca e che per questo motivo è meglio non tradurre al fine di garantirne un corretto utilizzo. Per questi motivi il documento non verrà tradotto. Per favore fate riferimento al documento originale in lingua inglese.

Ed infine, qui ci sono alcune guide più tecniche che son state messe qua solo perché non si è trovato un posto migliore.

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/applying-patches.rst

**Translator** Federico Vaga <[federico.vaga@vaga.pv.it](mailto:federico.vaga@vaga.pv.it)>

## Applicare patch al kernel Linux

---

**Note:** Questo documento è obsoleto. Nella maggior parte dei casi, piuttosto che usare patch manualmente, vorrete usare Git. Per questo motivo il documento non verrà tradotto.

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/adding-syscalls.rst

**Translator** Federico Vaga <[federico.vaga@vaga.pv.it](mailto:federico.vaga@vaga.pv.it)>

## Aggiungere una nuova chiamata di sistema

Questo documento descrive quello che è necessario sapere per aggiungere nuove chiamate di sistema al kernel Linux; questo è da considerarsi come un'aggiunta ai soliti consigli su come proporre nuove modifiche [Documentation/translations/it\\_IT/process/submitting-patches.rst](#).

## Alternative alle chiamate di sistema

La prima considerazione da fare quando si aggiunge una nuova chiamata di sistema è quella di valutare le alternative. Nonostante le chiamate di sistema siano il punto di interazione fra spazio utente e kernel più tradizionale ed ovvio, esistono altre possibilità - scegliete quella che meglio si adatta alle vostra interfaccia.

- Se le operazioni coinvolte possono rassomigliare a quelle di un filesystem, allora potrebbe avere molto più senso la creazione di un nuovo filesystem o dispositivo. Inoltre, questo rende più facile incapsulare la nuova funzionalità in un modulo kernel piuttosto che essere sviluppata nel cuore del kernel.
  - Se la nuova funzionalità prevede operazioni dove il kernel notifica lo spazio utente su un avvenimento, allora restituire un descrittore di file all'oggetto corrispondente permette allo spazio utente di utilizzare `poll/select/epoll` per ricevere quelle notifiche.
  - Tuttavia, le operazioni che non si sposano bene con operazioni tipo `read(2)/write(2)` dovrebbero essere implementate come chiamate `ioctl(2)`, il che potrebbe portare ad un'API in un qualche modo opaca.
- Se dovete esporre solo delle informazioni sul sistema, un nuovo nodo in sysfs (vedere [Documentation/filesystems/sysfs.rst](#)) o in procfs potrebbe essere sufficiente. Tuttavia, l'accesso a questi meccanismi richiede che il filesystem sia montato, il che potrebbe non essere sempre vero (per esempio, in ambienti come namespace/sandbox/chroot). Evitate d'aggiungere nuove API in debugfs perché questo non viene considerata un'interfaccia di 'produzione' verso lo spazio utente.
- Se l'operazione è specifica ad un particolare file o descrittore, allora potrebbe essere appropriata l'aggiunta di un comando `fcntl(2)`. Tuttavia, `fcntl(2)` è una chiamata di sistema multiplatrice che nasconde una notevole complessità, quindi è ottima solo quando la nuova funzione assomiglia a quelle già esistenti in `fcntl(2)`, oppure la nuova funzionalità è veramente semplice (per esempio, leggere/scrivere un semplice flag associato ad un descrittore di file).
- Se l'operazione è specifica ad un particolare processo, allora potrebbe essere appropriata l'aggiunta di un comando `prctl(2)`. Come per `fcntl(2)`, questa chiamata di sistema è un complesso multiplatore quindi è meglio usarlo per cose molto simili a quelle esistenti nel comando `prctl` oppure per leggere/scrivere un semplice flag relativo al processo.

## Progettare l'API: pianificare le estensioni

Una nuova chiamata di sistema diventerà parte dell'API del kernel, e dev'essere supportata per un periodo indefinito. Per questo, è davvero un'ottima idea quella di discutere apertamente l'interfaccia sulla lista di discussione del kernel, ed è altrettanto importante pianificarne eventuali estensioni future.

(Nella tabella delle chiamate di sistema sono disseminati esempi dove questo non fu fatto, assieme ai corrispondenti aggiornamenti - eventfd/eventfd2, dup2/dup3, inotify\_init/inotify\_init1, pipe/pipe2, renameat/renameat2 -quindi imparate dalla storia del kernel e pianificate le estensioni fin dall'inizio)

Per semplici chiamate di sistema che accettano solo un paio di argomenti, il modo migliore di permettere l'estensibilità è quello di includere un argomento *flags* alla chiamata di sistema. Per assicurarsi che i programmi dello spazio utente possano usare in sicurezza *flags* con diverse versioni del kernel, verificate se *flags* contiene un qualsiasi valore sconosciuto, in qual caso rifiutate la chiamata di sistema (con EINVAL):

```
if (flags & ~(THING_FLAG1 | THING_FLAG2 | THING_FLAG3))
    return -EINVAL;
```

(Se *flags* non viene ancora utilizzato, verificate che l'argomento sia zero)

Per chiamate di sistema più sofisticate che coinvolgono un numero più grande di argomenti, il modo migliore è quello di incapsularne la maggior parte in una struttura dati che verrà passata per puntatore. Questa struttura potrà funzionare con future estensioni includendo un campo *size*:

```
struct xyzzy_params {
    u32 size; /* userspace sets p->size = sizeof(struct xyzzy_params) */
    u32 param_1;
    u64 param_2;
    u64 param_3;
};
```

Fintanto che un qualsiasi campo nuovo, diciamo *param\_4*, è progettato per offrire il comportamento precedente quando vale zero, allora questo permetterà di gestire un conflitto di versione in entrambe le direzioni:

- un vecchio kernel può gestire l'accesso di una versione moderna di un programma in spazio utente verificando che la memoria oltre la dimensione della struttura dati attesa sia zero (in pratica verificare che *param\_4 == 0*).
- un nuovo kernel può gestire l'accesso di una versione vecchia di un programma in spazio utente estendendo la struttura dati con zeri (in pratica *param\_4 = 0*).

Vedere *perf\_event\_open(2)* e la funzione *perf\_copy\_attr()* (in *kernel/events/core.c*) per un esempio pratico di questo approccio.

## Progettare l'API: altre considerazioni

Se la vostra nuova chiamata di sistema permette allo spazio utente di fare riferimento ad un oggetto del kernel, allora questa dovrebbe usare un descrittore di file per accesso all'oggetto - non inventatevi nuovi tipi di accesso da spazio utente quando il kernel ha già dei meccanismi e una semantica ben definita per utilizzare i descrittori di file.

Se la vostra nuova chiamata di sistema `xyzzy(2)` ritorna un nuovo descrittore di file, allora l'argomento `flags` dovrebbe includere un valore equivalente a `O_CLOEXEC` per i nuovi descrittori. Questo rende possibile, nello spazio utente, la chiusura della finestra temporale fra le chiamate a `xyzzy()` e `fcntl(fd, F_SETFD, FD_CLOEXEC)`, dove un inaspettato `fork()` o `execve()` potrebbe trasferire il descrittore al programma eseguito (Comunque, resistete alla tentazione di riutilizzare il valore di `O_CLOEXEC` dato che è specifico dell'architettura e fa parte di una enumerazione di flag `O_*` che è abbastanza ricca).

Se la vostra nuova chiamata di sistema ritorna un nuovo descrittore di file, dovreste considerare che significato avrà l'uso delle chiamate di sistema della famiglia di `poll(2)`. Rendere un descrittore di file pronto per la lettura o la scrittura è il tipico modo del kernel per notificare lo spazio utente circa un evento associato all'oggetto del kernel.

Se la vostra nuova chiamata di sistema `xyzzy(2)` ha un argomento che è il percorso ad un file:

```
int sys_xyzzy(const char __user *path, ..., unsigned int flags);
```

dovreste anche considerare se non sia più appropriata una versione `xyzzyat(2)`:

```
int sys_xzzyat(int dfd, const char __user *path, ..., unsigned int flags);
```

Questo permette più flessibilità su come lo spazio utente specificherà il file in questione; in particolare, permette allo spazio utente di richiedere la funzionalità su un descrittore di file già aperto utilizzando il flag `AT_EMPTY_PATH`, in pratica otterremmo gratuitamente l'operazione `fxyzzy(3)`:

- `xzzyat(AT_FDCWD, path, ..., 0)` is equivalent to `xyzzy(path, ...)`
- `xzzyat(fd, "", ..., AT_EMPTY_PATH)` is equivalent to `fxyzzy(fd, ...)`

(Per maggiori dettagli sulla logica delle chiamate `*at()`, leggete la pagina man `openat(2)`; per un esempio di `AT_EMPTY_PATH`, leggere la pagina man `fstatat(2)`).

Se la vostra nuova chiamata di sistema `xyzzy(2)` prevede un parametro per descrivere uno scostamento all'interno di un file, usate `loff_t` come tipo cosicché scostamenti a 64-bit potranno essere supportati anche su architetture a 32-bit.

Se la vostra nuova chiamata di sistema `xyzzy(2)` prevede l'uso di funzioni riservate, allora dev'essere gestita da un opportuno bit di privilegio (verificato con una chiamata a `capable()`), come descritto nella pagina man `capabilities(7)`. Scegliete un bit di privilegio già esistente per gestire la funzionalità associata, ma evitate la combinazione di diverse funzionalità vagamente collegate dietro lo stesso bit, in quanto va contro il principio di `capabilities` di separare i poteri di root. In particolare, evitate di aggiungere nuovi usi al fin-troppo-generico privilegio `CAP_SYS_ADMIN`.

Se la vostra nuova chiamata di sistema `xyzzy(2)` manipola altri processi oltre a quello chiamato, allora dovrebbe essere limitata (usando la chiamata `ptrace_may_access()`) di modo che solo un processo chiamante con gli stessi permessi del processo in oggetto, o con i necessari privilegi, possa manipolarlo.

Infine, state attenti che in alcune architetture non-x86 la vita delle chiamate di sistema con argomenti a 64-bit viene semplificata se questi argomenti ricadono in posizioni dispari (pratica, i parametri 1, 3, 5); questo permette l'uso di coppie contigue di registri a 32-bit. (Questo non conta se gli argomenti sono parte di una struttura dati che viene passata per puntatore).

### Proporre l'API

Al fine di rendere le nuove chiamate di sistema di facile revisione, è meglio che dividiate le modifiche i pezzi separati. Questi dovrebbero includere almeno le seguenti voci in *commit* distinti (ognuno dei quali sarà descritto più avanti):

- l'essenza dell'implementazione della chiamata di sistema, con i prototipi, i numeri generici, le modifiche al Kconfig e l'implementazione *stub* di ripiego.
- preparare la nuova chiamata di sistema per un'architettura specifica, solitamente x86 (ovvero tutti: x86\_64, x86\_32 e x32).
- un programma di auto-verifica da mettere in `tools/testing/selftests/` che mostri l'uso della chiamata di sistema.
- una bozza di pagina man per la nuova chiamata di sistema. Può essere scritta nell'email di presentazione, oppure come modifica vera e propria al repository delle pagine man.

Le proposte di nuove chiamate di sistema, come ogni altro modifica all'API del kernel, deve essere sottomessa alla lista di discussione [linux-api@vger.kernel.org](mailto:linux-api@vger.kernel.org).

### Implementazione di chiamate di sistema generiche

Il principale punto d'accesso alla vostra nuova chiamata di sistema `xyzzy(2)` verrà chiamato `sys_xyzzy()`; ma, piuttosto che in modo esplicito, lo aggiungerete tramite la macro `SYSCALL_DEFINEn`. La 'n' indica il numero di argomenti della chiamata di sistema; la macro ha come argomento il nome della chiamata di sistema, seguito dalle coppie (tipo, nome) per definire i suoi parametri. L'uso di questa macro permette di avere i metadati della nuova chiamata di sistema disponibili anche per altri strumenti.

Il nuovo punto d'accesso necessita anche del suo prototipo di funzione in `include/linux/syscalls.h`, marcato come `asmlinkage` di modo da abbinargli il modo in cui quelle chiamate di sistema verranno invocate:

```
asmlinkage long sys_xyzzy(...);
```

Alcune architetture (per esempio x86) hanno le loro specifiche tabelle di chiamate di sistema (`syscall`), ma molte altre architetture condividono una tabella comune di `syscall`. Aggiungete alla lista generica la vostra nuova chiamata di sistema aggiungendo un nuovo elemento alla lista in `include/uapi/asm-generic/unistd.h`:

```
#define __NR_xyzzy 292
__SYSCALL(__NR_xyzzy, sys_xyzzy)
```

Aggiornate anche il contatore `_NR_syscalls` di modo che sia coerente con l'aggiunta della nuove chiamate di sistema; va notato che se più di una nuova chiamata di sistema viene aggiunta nella stessa finestra di sviluppo, il numero della vostra nuova `syscall` potrebbe essere aggiustato al fine di risolvere i conflitti.

Il file `kernel/sys_ni.c` fornisce le implementazioni *stub* di ripiego che ritornano -ENOSYS. Aggiungete la vostra nuova chiamata di sistema anche qui:

```
COND_SYSCALL(xyzzy);
```

La vostra nuova funzionalità del kernel, e la chiamata di sistema che la controlla, dovrebbero essere opzionali. Quindi, aggiungete un'opzione CONFIG (solitamente in `init/Kconfig`). Come al solito per le nuove opzioni CONFIG:

- Includete una descrizione della nuova funzionalità e della chiamata di sistema che la controlla.
- Rendete l'opzione dipendente da EXPERT se dev'essere nascosta agli utenti normali.
- Nel Makefile, rendere tutti i nuovi file sorgenti, che implementano la nuova funzionalità, dipendenti dall'opzione CONFIG (per esempio `obj-$(CONFIG_XYZZY_SYSCALL) += xyzzy.o`).
- Controllate due volte che sia possibile generare il kernel con la nuova opzione CONFIG disabilitata.

Per riassumere, vi serve un *commit* che includa:

- un'opzione CONFIG`` per la nuova funzione, normalmente in ```init/Kconfig`
- `SYSCALL_DEFINEx(xyzzy, ...)` per il punto d'accesso
- il corrispondente prototipo in `include/linux/syscalls.h`
- un elemento nella tabella generica in `include/uapi/asm-generic/unistd.h`
- *stub* di ripiego in `kernel/sys_ni.c`

## Implementazione delle chiamate di sistema x86

Per collegare la vostra nuova chiamate di sistema alle piattaforme x86, dovete aggiornate la tabella principale di syscall. Assumendo che la vostra nuova chiamata di sistema non sia particolarmente speciale (vedere sotto), dovete aggiungere un elemento *common* (per x86\_64 e x32) in `arch/x86/entry/syscalls/syscall_64.tbl`:

```
333    common    xyzzy    sys_xyzzy
```

e un elemento per *i386* `arch/x86/entry/syscalls/syscall_32.tbl`:

```
380    i386    xyzzy    sys_xyzzy
```

Ancora una volta, questi numeri potrebbero essere cambiati se generano conflitti durante la finestra di integrazione.

## Chiamate di sistema compatibili (generico)

Per molte chiamate di sistema, la stessa implementazione a 64-bit può essere invocata anche quando il programma in spazio utente è a 32-bit; anche se la chiamata di sistema include esplicitamente un puntatore, questo viene gestito in modo trasparente.

Tuttavia, ci sono un paio di situazioni dove diventa necessario avere un livello di gestione della compatibilità per risolvere le differenze di dimensioni fra 32-bit e 64-bit.

Il primo caso è quando un kernel a 64-bit supporta anche programmi in spazio utente a 32-bit, perciò dovrà ispezionare aree della memoria (`_user`) che potrebbero contenere valori a 32-bit o a 64-bit. In particolar modo, questo è necessario quando un argomento di una chiamata di sistema è:

- un puntatore ad un puntatore
- un puntatore ad una struttura dati contenente a sua volta un puntatore (ad esempio `struct iovec __user *`)
- un puntatore ad un tipo intero di dimensione variabile (`time_t`, `off_t`, `long`, ...)
- un puntatore ad una struttura dati contenente un tipo intero di dimensione variabile.

Il secondo caso che richiede un livello di gestione della compatibilità è quando uno degli argomenti di una chiamata a sistema è esplicitamente un tipo a 64-bit anche su architetture a 32-bit, per esempio `loff_t` o `__u64`. In questo caso, un valore che arriva ad un kernel a 64-bit da un'applicazione a 32-bit verrà diviso in due valori a 32-bit che dovranno essere riassemblati in questo livello di compatibilità.

(Da notare che non serve questo livello di compatibilità per argomenti che sono puntatori ad un tipo esplicitamente a 64-bit; per esempio, in `splice(2)` l'argomento di tipo `loff_t __user *` non necessita di una chiamata di sistema `compat_`)

La versione compatibile della nostra chiamata di sistema si chiamerà `compat_sys_xzzy()`, e viene aggiunta utilizzando la macro `COMPAT_SYSCALL_DEFINEn()` (simile a `SYSCALL_DEFINEn`). Questa versione dell'implementazione è parte del kernel a 64-bit ma accetta parametri a 32-bit che trasformerà secondo le necessità (tipicamente, la versione `compat_sys_` converte questi valori nello loro corrispondente a 64-bit e può chiamare la versione `sys_` oppure invocare una funzione che implementa le parti comuni).

Il punto d'accesso `compat` deve avere il corrispondente prototipo di funzione in `include/linux/compat.h`, marcato come asmlinkage di modo da abbinargli il modo in cui quelle chiamate di sistema verranno invocate:

```
asmlinkage long compat_sys_xzzy(...);
```

Se la chiamata di sistema prevede una struttura dati organizzata in modo diverso per sistemi a 32-bit e per quelli a 64-bit, diciamo `struct xyzzy_args`, allora il file d'intestazione `then the include/linux/compat.h` deve includere la sua versione `compatibile` (`struct compat_xyzzy_args`); ogni variabile con dimensione variabile deve avere il proprio tipo `compat_` corrispondente a quello in `struct xyzzy_args`. La funzione `compat_sys_xzzy()` può usare la struttura `compat_` per analizzare gli argomenti ricevuti da una chiamata a 32-bit.

Per esempio, se avete i seguenti campi:

```
struct xyzzy_args {
    const char __user *ptr;
```

```
__kernel_long_t varying_val;
u64 fixed_val;
/* ... */
};
```

nella struttura `struct xyzzy_args`, allora la struttura `struct compat_xyzzy_args` dovrebbe avere:

```
struct compat_xyzzy_args {
    compat_uptr_t ptr;
    compat_long_t varying_val;
    u64 fixed_val;
    /* ... */
};
```

La lista generica delle chiamate di sistema ha bisogno di essere aggiustata al fine di permettere l'uso della versione *compatibile*; la voce in `include/uapi/asm-generic/unistd.h` dovrebbero usare `_SC_COMP` piuttosto di `_SYSCALL`:

```
#define __NR_xyzzy 292
__SC_COMP(__NR_xyzzy, sys_xyzzy, compat_sys_xyzzy)
```

Riassumendo, vi serve:

- un `COMPAT_SYSCALL_DEFINEx`(`xyzzy`, ...) per il punto d'accesso *compatibile*
- un prototipo in `include/linux/compat.h`
- (se necessario) una struttura di compatibilità a 32-bit in `include/linux/compat.h`
- una voce `_SC_COMP`, e non `_SYSCALL`, in `include/uapi/asm-generic/unistd.h`

## Compatibilità delle chiamate di sistema (x86)

Per collegare una chiamata di sistema, su un'architettura x86, con la sua versione *compatibile*, è necessario aggiustare la voce nella tabella delle syscall.

Per prima cosa, la voce in `arch/x86/entry/syscalls/syscall_32.tbl` prende un argomento aggiuntivo per indicare che un programma in spazio utente a 32-bit, eseguito su un kernel a 64-bit, dovrebbe accedere tramite il punto d'accesso compatibile:

380	i386	xyzzy	sys_xyzzy	<code>_ia32_compat_sys_xyzzy</code>
-----	------	-------	-----------	-------------------------------------

Secondo, dovete capire cosa dovrebbe succedere alla nuova chiamata di sistema per la versione dell'ABI x32. Qui C'è una scelta da fare: gli argomenti possono corrispondere alla versione a 64-bit o a quella a 32-bit.

Se c'è un puntatore ad un puntatore, la decisione è semplice: x32 è ILP32, quindi gli argomenti dovrebbero corrispondere a quelli a 32-bit, e la voce in `arch/x86/entry/syscalls/syscall_64.tbl` sarà divisa cosicché i programmi x32 eseguano la chiamata *compatibile*:

333	64	xyzzy	sys_xyzzy
...			
555	x32	xyzzy	<code>_x32_compat_sys_xyzzy</code>

Se non ci sono puntatori, allora è preferibile riutilizzare la chiamata di sistema a 64-bit per l'ABI x32 (e di conseguenza la voce in arch/x86/entry/syscalls/syscall\_64.tbl rimane immutata).

In ambo i casi, dovreste verificare che i tipi usati dagli argomenti abbiano un'esatta corrispondenza da x32 (-mx32) al loro equivalente a 32-bit (-m32) o 64-bit (-m64).

### Chiamate di sistema che ritornano altrove

Nella maggior parte delle chiamate di sistema, al termine della loro esecuzione, i programmi in spazio utente riprendono esattamente dal punto in cui si erano interrotti – quindi dall'istruzione successiva, con lo stesso *stack* e con la maggior parte dei registri com'erano stati lasciati prima della chiamata di sistema, e anche con la stessa memoria virtuale.

Tuttavia, alcune chiamata di sistema fanno le cose in modo differente. Potrebbero ritornare ad un punto diverso (*rt\_sigreturn*) o cambiare la memoria in spazio utente (*fork/vfork/clone*) o perfino l'architettura del programma (*execve/execveat*).

Per permettere tutto ciò, l'implementazione nel kernel di questo tipo di chiamate di sistema potrebbero dover salvare e ripristinare registri aggiuntivi nello *stack* del kernel, permettendo così un controllo completo su dove e come l'esecuzione dovrà continuare dopo l'esecuzione della chiamata di sistema.

Queste saranno specifiche per ogni architettura, ma tipicamente si definiscono dei punti d'accesso in *assembly* per salvare/ripristinare i registri aggiuntivi e quindi chiamare il vero punto d'accesso per la chiamata di sistema.

Per l'architettura x86\_64, questo è implementato come un punto d'accesso *stub\_xyzzy* in arch/x86/entry/entry\_64.S, e la voce nella tabella di syscall (arch/x86/entry/syscalls/syscall\_64.tbl) verrà corretta di conseguenza:

```
333    common    xyzzy    stub_xyzzy
```

L'equivalente per programmi a 32-bit eseguiti su un kernel a 64-bit viene normalmente chiamato *stub32\_xyzzy* e implementato in arch/x86/entry/entry\_64\_compatible.S con la corrispondente voce nella tabella di syscall arch/x86/entry/syscalls/syscall\_32.tbl corretta nel seguente modo:

```
380    i386    xyzzy    sys_xyzzy    stub32_xyzzy
```

Se una chiamata di sistema necessita di un livello di compatibilità (come nella sezione precedente), allora la versione *stub32\_* deve invocare la versione *compat\_sys\_* piuttosto che quella nativa a 64-bit. In aggiunta, se l'implementazione dell'ABI x32 è diversa da quella x86\_64, allora la sua voce nella tabella di syscall dovrà chiamare uno *stub* che invoca la versione *compat\_sys\_*.

Per completezza, sarebbe carino impostare una mappatura cosicché *user-mode* Linux (UML) continui a funzionare – la sua tabella di syscall farà riferimento a *stub\_xyzzy*, ma UML non include l'implementazione in arch/x86/entry/entry\_64.S (perché UML simula i registri eccetera). Correggerlo è semplice, basta aggiungere una #define in arch/x86/um/sys\_call\_table\_64.c:

```
#define stub_xyzzy sys_xyzzy
```

## Altri dettagli

La maggior parte dei kernel tratta le chiamate di sistema allo stesso modo, ma possono esserci rare eccezioni per le quali potrebbe essere necessario l'aggiornamento della vostra chiamata di sistema.

Il sotto-sistema di controllo (*audit subsystem*) è uno di questi casi speciali; esso include (per architettura) funzioni che classificano alcuni tipi di chiamate di sistema - in particolare apertura dei file (`open/openat`), esecuzione dei programmi (`execve/exeveat`) oppure multipliatori di socket (`socketcall`). Se la vostra nuova chiamata di sistema è simile ad una di queste, allora il sistema di controllo dovrebbe essere aggiornato.

Più in generale, se esiste una chiamata di sistema che è simile alla vostra, vale la pena fare una ricerca con `grep` su tutto il kernel per la chiamata di sistema esistente per verificare che non ci siano altri casi speciali.

## Verifica

Una nuova chiamata di sistema dev'essere, ovviamente, provata; è utile fornire ai revisori un programma in spazio utente che mostri l'uso della chiamata di sistema. Un buon modo per combinare queste cose è quello di aggiungere un semplice programma di auto-verifica in una nuova cartella in `tools/testing/selftests/`.

Per una nuova chiamata di sistema, ovviamente, non ci sarà alcuna funzione in `libc` e quindi il programma di verifica dovrà invocarla usando `syscall()`; inoltre, se la nuova chiamata di sistema prevede un nuova struttura dati visibile in spazio utente, il file d'intestazione necessario dev'essere installato al fine di compilare il programma.

Assicuratevi che il programma di auto-verifica possa essere eseguito correttamente su tutte le architetture supportate. Per esempio, verificate che funzioni quando viene compilato per `x86_64 (-m64)`, `x86_32 (-m32)` e `x32 (-mx32)`.

Al fine di una più meticolosa ed estesa verifica della nuova funzionalità, dovreste considerare l'aggiunta di nuove verifica al progetto 'Linux Test', oppure al progetto `xfstests` per cambiamenti relativi al filesystem.

- <https://linux-test-project.github.io/>
- <git://git.kernel.org/pub/scm/fs/xfs/xfstests-dev.git>

## Pagine man

Tutte le nuove chiamate di sistema dovrebbero avere una pagina man completa, idealmente usando i marcatori `groff`, ma anche il puro testo può andare. Se state usando `groff`, è utile che includiate nella email di presentazione una versione già convertita in formato ASCII: semplificherà la vita dei revisori.

Le pagine man dovrebbero essere in copia-conoscenza verso [linux-man@vger.kernel.org](mailto:linux-man@vger.kernel.org) Per maggiori dettagli, leggere <https://www.kernel.org/doc/man-pages/patches.html>

### Non invocate chiamate di sistema dal kernel

Le chiamate di sistema sono, come già detto prima, punti di interazione fra lo spazio utente e il kernel. Perciò, le chiamate di sistema come `sys_xyzzy()` o `compat_sys_xyzzy()` dovrebbero essere chiamate solo dallo spazio utente attraverso la tabella `syscall`, ma non da nessun altro punto nel kernel. Se la nuova funzionalità è utile all'interno del kernel, per esempio dev'essere condivisa fra una vecchia e una nuova chiamata di sistema o dev'essere utilizzata da una chiamata di sistema e la sua variante compatibile, allora dev'essere implementata come una funzione di supporto (*helper function*) (per esempio `ksys_xyzzy()`). Questa funzione potrà essere chiamata dallo *stub* (`sys_xyzzy()`), dalla variante compatibile (`compat_sys_xyzzy()`), e/o da altri parti del kernel.

Sui sistemi x86 a 64-bit, a partire dalla versione v4.17 è un requisito fondamentale quello di non invocare chiamate di sistema all'interno del kernel. Esso usa una diversa convenzione per l'invocazione di chiamate di sistema dove `struct pt_regs` viene decodificata al volo in una funzione che racchiude la chiamata di sistema la quale verrà eseguita successivamente. Questo significa che verranno passati solo i parametri che sono davvero necessari ad una specifica chiamata di sistema, invece che riempire ogni volta 6 registri del processore con contenuti presi dallo spazio utente (potrebbe causare seri problemi nella sequenza di chiamate).

Inoltre, le regole su come i dati possano essere usati potrebbero differire fra il kernel e l'utente. Questo è un altro motivo per cui invocare `sys_xyzzy()` è generalmente una brutta idea.

Eccezioni a questa regola vengono accettate solo per funzioni d'architetture che surclassano quelle generiche, per funzioni d'architettura di compatibilità, o per altro codice in arch/

### Riferimenti e fonti

- Articolo di Michael Kerris su LWN sull'uso dell'argomento flags nelle chiamate di sistema: <https://lwn.net/Articles/585415/>
- Articolo di Michael Kerris su LWN su come gestire flag sconosciuti in una chiamata di sistema: <https://lwn.net/Articles/588444/>
- Articolo di Jake Edge su LWN che descrive i limiti degli argomenti a 64-bit delle chiamate di sistema: <https://lwn.net/Articles/311630/>
- Una coppia di articoli di David Drysdale che descrivono i dettagli del percorso implementativo di una chiamata di sistema per la versione v3.14:
  - <https://lwn.net/Articles/604287/>
  - <https://lwn.net/Articles/604515/>
- Requisiti specifici alle architetture sono discussi nella pagina man `syscall(2)` : <http://man7.org/linux/man-pages/man2/syscall.2.html#NOTES>
- Collezione di email di Linux Torvalds sui problemi relativi a `ioctl()`: <http://yarchive.net/comp/linux/ioctl.html>
- “Come non inventare interfacce del kernel”, Arnd Bergmann, <http://www.ukuug.org/events/linux2007/2007/papers/Bergmann.pdf>
- Articolo di Michael Kerris su LWN sull'evitare nuovi usi di CAP\_SYS\_ADMIN: <https://lwn.net/Articles/486306/>

- Raccomandazioni da Andrew Morton circa il fatto che tutte le informazioni su una nuova chiamata di sistema dovrebbero essere contenute nello stesso filone di discussione di email: <https://lore.kernel.org/r/20140724144747.3041b208832bbdf9fbce5d96@linux-foundation.org>
- Raccomandazioni da Michael Kerrisk circa il fatto che le nuove chiamate di sistema dovrebbero avere una pagina man: <https://lore.kernel.org/r/CAKgNAkgMA39AfoSoA5Pe1r9N+ZzFYQNvNPvcRN7tOvRb8+v06Q@mail.gmail.com>
- Consigli da Thomas Gleixner sul fatto che il collegamento all'architettura x86 dovrebbe avvenire in un *commit* differente: <https://lore.kernel.org/r/alpine.DEB.2.11.1411191249560.3909@nanos>
- Consigli da Greg Kroah-Hartman circa la bontà d'avere una pagina man e un programma di auto-verifica per le nuove chiamate di sistema: <https://lore.kernel.org/r/20140320025530.GA25469@kroah.com>
- Discussione di Michael Kerrisk sulle nuove chiamate di sistema contro le estensioni *prctl(2)*: <https://lore.kernel.org/r/CAHO5Pa3F2MjfTtfNxa8LbnkeeU8=YJ+9tDqxZpw7Gz59E-4AUg@mail.gmail.com>
- Consigli da Ingo Molnar che le chiamate di sistema con più argomenti dovrebbero incapsularli in una struttura che includa un argomento *size* per garantire l'estensibilità futura: <https://lore.kernel.org/r/20150730083831.GA22182@gmail.com>
- Un certo numero di casi strani emersi dall'uso (riuso) dei flag O\_ \*:
  - commit 75069f2b5bfb ("vfs: renumber FMODE\_NONOTIFY and add to uniqueness check")
  - commit 12ed2e36c98a ("fanotify: FMODE\_NONOTIFY and \_\_O\_SYNC in sparc conflict")
  - commit bb458c644a59 ("Safer ABI for O\_TMPFILE")
- Discussion from Matthew Wilcox about restrictions on 64-bit arguments: <https://lore.kernel.org/r/20081212152929.GM26095@parisc-linux.org>
- Raccomandazioni da Greg Kroah-Hartman sul fatto che i flag sconosciuti dovrebbero essere controllati: <https://lore.kernel.org/r/20140717193330.GB4703@kroah.com>
- Raccomandazioni da Linus Torvalds che le chiamate di sistema x32 dovrebbero favorire la compatibilità con le versioni a 64-bit piuttosto che quelle a 32-bit: <https://lore.kernel.org/r/CA+55aFxfmwfB7jbbrXxa=K7VBYPfAvmu3XOkGrLbB1UFjX1+Ew@mail.gmail.com>

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/magic-number.rst

**Translator** Federico Vaga <[federico.vaga@vaga.pv.it](mailto:federico.vaga@vaga.pv.it)>

### I numeri magici di Linux

Questo documento è un registro dei numeri magici in uso. Quando aggiungete un numero magico ad una struttura, dovreste aggiungerlo anche a questo documento; la cosa migliore è che tutti i numeri magici usati dalle varie strutture siano unici.

È **davvero** un'ottima idea proteggere le strutture dati del kernel con dei numeri magici. Questo vi permette in fase d'esecuzione di (a) verificare se una struttura è stata malmenata, o (b) avete passato a una procedura la struttura errata. Quest'ultimo è molto utile - particolarmente quando si passa una struttura dati tramite un puntatore void \*. Il codice tty, per esempio, effettua questa operazione con regolarità passando avanti e indietro le strutture specifiche per driver e discipline.

Per utilizzare un numero magico, dovete dichiararlo all'inizio della struttura dati, come di seguito:

```
struct tty_ldisc {  
    int      magic;  
    ...  
};
```

Per favore, seguite questa direttiva quando aggiungerete migliorie al kernel! Mi ha risparmiato un numero illimitato di ore di debug, specialmente nei casi più ostici dove si è andati oltre la dimensione di un vettore e la struttura dati che lo seguiva in memoria è stata sovrascritta. Seguendo questa direttiva, questi casi vengono identificati velocemente e in sicurezza.

Registro dei cambiamenti:

Theodore Ts'o  
31 Mar 94

La tabella magica è aggiornata a Linux 2.1.55.

Michael Chastain  
<mailto:mec@shout.net>  
22 Sep 1997

Ora dovrebbe essere aggiornata a Linux 2.1.112. Dato che siamo in un momento di congelamento delle funzionalità (\*feature freeze\*) è improbabile che qualcosa cambi prima della versione 2.2.x. Le righe sono ordinate secondo il campo numero.

Krzysztof G. Baranowski  
<mailto: kgb@knm.org.pl>  
29 Jul 1998

Aggiornamento della tabella a Linux 2.5.45. Giusti nel congelamento delle funzionalità ma è comunque possibile che qualche nuovo numero magico s'intrufoli prima del kernel 2.6.x.

Petr Baudis  
<pasky@ucw.cz>

03 Nov 2002

Aggiornamento della tabella magica a Linux 2.5.74.

Fabian Frederick  
 <ffrederick@users.sourceforge.net>  
 09 Jul 2003

Nome magico	Numero	Struttura	File
PG_MAGIC	'P'	pg_{read,write}_hdr	include/linux/pg.h
CMAGIC	0x0111	user	include/linux/a.c
MKISS_DRIVER_MAGIC	0x04bf	mkiss_channel	drivers/net/mkiss.c
HDLC_MAGIC	0x239e	n_hdlc	drivers/char/n_hdlc.c
APM BIOS MAGIC	0x4101	apm_user	arch/x86/kernel/apm.c
DB_MAGIC	0x4442	fc_info	drivers/net/iph552.c
DL_MAGIC	0x444d	fc_info	drivers/net/iph55d.c
FASYNC_MAGIC	0x4601	fasync_struct	include/linux/fs.h
FF_MAGIC	0x4646	fc_info	drivers/net/iph55f.c
PTY_MAGIC	0x5001		drivers/char/pty.c
PPP_MAGIC	0x5002	ppp	include/linux/if_ether.h
SSTATE_MAGIC	0x5302	serial_state	include/linux/serial.h
SLIP_MAGIC	0x5302	slip	drivers/net/slip.c
STRIP_MAGIC	0x5303	strip	drivers/net/strip.c
SIXPACK_MAGIC	0x5304	sixpack	drivers/net/hamradio.c
AX25_MAGIC	0x5316	ax_disp	drivers/net/mkiss.c
TTY_MAGIC	0x5401	tty_struct	include/linux/tty.h
MGSL_MAGIC	0x5401	mgsl_info	drivers/char/sync.c
TTY_DRIVER_MAGIC	0x5402	tty_driver	include/linux/tty.h
MGSLPC_MAGIC	0x5402	mgslpc_info	drivers/char/pcmcia.c
USB_SERIAL_MAGIC	0x6702	usb_serial	drivers/usb/serial.c
FULL_DUPLEX_MAGIC	0x6969		drivers/net/ether.c
USB_BLUETOOTH_MAGIC	0x6d02	usb_bluetooth	drivers/usb/class.c
RFCOMM_TTY_MAGIC	0x6d02		net/bluetooth/rfcomm.c
USB_SERIAL_PORT_MAGIC	0x7301	usb_serial_port	drivers/usb/serial.c
CG_MAGIC	0x00090255	ufs_cylinder_group	include/linux/ufs.h
LSEMAGIC	0x05091998	lse	drivers/fc4/fc.c
RIEBL_MAGIC	0x09051990		drivers/net/atari.c
NBD_REQUEST_MAGIC	0x12560953	nbd_request	include/linux/nbd.h
RED_MAGIC2	0x170fc2a5	(any)	mm/slab.c
BAYCOM_MAGIC	0x19730510	baycom_state	drivers/net/baycom.c
ISDN_X25IFACE_MAGIC	0x1e75a2b9	isdn_x25iface_proto_data	drivers/isdn/isdn-x25.c
ECP_MAGIC	0x21504345	cdkecpsig	include/linux/cdkepsig.h
LSOMAGIC	0x27091997	lso	drivers/fc4/fc.c
LSMAGIC	0x2a3b4d2a	ls	drivers/fc4/fc.c
WANPIPE_MAGIC	0x414C4453	sdla_{dump,exec}	include/linux/wanpipe.h
CS_CARD_MAGIC	0x43525553	cs_card	sound/oss/cs46xx.c
LABELCL_MAGIC	0x4857434c	labelcl_info_s	include/asm/ia64/include/labelcl.h
ISDN_ASYNC_MAGIC	0x49344C01	modem_info	include/linux/isdn.h

Table 1 – continued from previous page

Nome magico	Numero	Struttura	File
CTC_ASYNC_MAGIC	0x49344C01	ctc_tty_info	drivers/s390/net/
ISDN_NET_MAGIC	0x49344C02	isdn_net_local_s	drivers/isdn/i4l/
SAVEKMSG_MAGIC2	0x4B4D5347	savekmsg	arch/*/amiga/conf
CS_STATE_MAGIC	0x4c4f4749	cs_state	sound/oss/cs46xx.
SLAB_C_MAGIC	0x4f17a36d	kmem_cache	mm/slab.c
COW_MAGIC	0x4f4f4f4d	cow_header_v1	arch/um/drivers/u
I810_CARD_MAGIC	0x5072696E	i810_card	sound/oss/i810_a
TRIDENT_CARD_MAGIC	0x5072696E	trident_card	sound/oss/trident
ROUTER_MAGIC	0x524d4157	wan_device	[in wanrouter.h pre
SAVEKMSG_MAGIC1	0x53415645	savekmsg	arch/*/amiga/conf
GDA_MAGIC	0x58464552	gda	arch/mips/include
RED_MAGIC1	0x5a2cf071	(any)	mm/slab.c
EEPROM_MAGIC_VALUE	0x5ab478d2	lanai_dev	drivers/atm/lanai
HDLCDRV_MAGIC	0x5ac6e778	hdlcdrv_state	include/linux/hdl
PCXX_MAGIC	0x5c6df104	channel	drivers/char/pcxx
KV_MAGIC	0x5f4b565f	kernel_vars_s	arch/mips/include
I810_STATE_MAGIC	0x63657373	i810_state	sound/oss/i810_a
TRIDENT_STATE_MAGIC	0x63657373	trient_state	sound/oss/trident
M3_CARD_MAGIC	0x646e6f50	m3_card	sound/oss/maestr
FW_HEADER_MAGIC	0x65726F66	fw_header	drivers/atm/fore2
SLOT_MAGIC	0x67267321	slot	drivers/hotplug/c
SLOT_MAGIC	0x67267322	slot	drivers/hotplug/a
LO_MAGIC	0x68797548	nbd_device	include/linux/nbc
M3_STATE_MAGIC	0x734d724d	m3_state	sound/oss/maestr
VMALLOC_MAGIC	0x87654320	snd_alloc_track	sound/core/memory
KMALLOC_MAGIC	0x87654321	snd_alloc_track	sound/core/memory
PWC_MAGIC	0x89DC10AB	pwc_device	drivers/usb/media
NBD_REPLY_MAGIC	0x96744668	nbd_reply	include/linux/nbc
ENI155_MAGIC	0xa54b872d	midway_eeprom	drivers/atm/eni.h
CODA_MAGIC	0xC0DAC0DA	coda_file_info	fs/coda/coda_fs_i
YAM_MAGIC	0xF10A7654	yam_port	drivers/net/hamra
CCB_MAGIC	0xf2691ad2	ccb	drivers/scsi/ncr5
QUEUE_MAGIC_FREE	0xf7e1c9a3	queue_entry	drivers/scsi/arm/
QUEUE_MAGIC_USED	0xf7e1cc33	queue_entry	drivers/scsi/arm/
HTB_CMAGIC	0xFEFAFEF1	htb_class	net/sched/sch_htb
NMI_MAGIC	0x48414d4d455201	nmi_s	arch/mips/include

Da notare che ci sono anche dei numeri magici specifici per driver nel *sound memory management*. Consultate `include/sound/sndmagic.h` per una lista completa. Molti driver audio OSS hanno i loro numeri magici costruiti a partire dall'identificativo PCI della scheda audio - nemmeno questi sono elencati in questo file.

Il file-system HFS è un altro grande utilizzatore di numeri magici - potete trovarli qui `fs/hfs/hfs.h`.

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni con-

sultate le [avvertenze](#).

**Original** Documentation/process/volatile-considered-harmful.rst

**Translator** Federico Vaga <[federico.vaga@vaga.pv.it](mailto:federico.vaga@vaga.pv.it)>

## Perché la parola chiave “volatile” non dovrebbe essere usata

Spesso i programmatori C considerano volatili quelle variabili che potrebbero essere cambiate al di fuori dal thread di esecuzione corrente; come risultato, a volte saranno tentati dall'utilizzare *volatile* nel kernel per le strutture dati condivise. In altre parole, gli è stato insegnato ad usare *volatile* come una variabile atomica di facile utilizzo, ma non è così. L'uso di *volatile* nel kernel non è quasi mai corretto; questo documento ne descrive le ragioni.

Il punto chiave da capire su *volatile* è che il suo scopo è quello di sopprimere le ottimizzazioni, che non è quasi mai quello che si vuole. Nel kernel si devono proteggere le strutture dati condivise contro accessi concorrenti e indesiderati: questa è un'attività completamente diversa. Il processo di protezione contro gli accessi concorrenti indesiderati eviterà anche la maggior parte dei problemi relativi all'ottimizzazione in modo più efficiente.

Come *volatile*, le primitive del kernel che rendono sicuro l'accesso ai dati (spinlock, mutex, barriere di sincronizzazione, ecc) sono progettate per prevenire le ottimizzazioni indesiderate. Se vengono usate opportunamente, non ci sarà bisogno di utilizzare *volatile*. Se vi sembra che *volatile* sia comunque necessario, ci dev'essere quasi sicuramente un baco da qualche parte. In un pezzo di codice kernel scritto a dovere, *volatile* può solo servire a rallentare le cose.

Considerate questo tipico blocco di codice kernel:

```
spin_lock(&the_lock);
do_something_on(&shared_data);
do_something_else_with(&shared_data);
spin_unlock(&the_lock);
```

Se tutto il codice seguisse le regole di sincronizzazione, il valore di un dato condiviso non potrebbe cambiare inaspettatamente mentre si trattiene un lock. Un qualsiasi altro blocco di codice che vorrà usare quel dato rimarrà in attesa del lock. Gli spinlock agiscono come barriere di sincronizzazione - sono stati esplicitamente scritti per agire così - il che significa che gli accessi al dato condiviso non saranno ottimizzati. Quindi il compilatore potrebbe pensare di sapere cosa ci sarà nel dato condiviso ma la chiamata `spin_lock()`, che agisce come una barriera di sincronizzazione, gli imporrà di dimenticarsi tutto ciò che sapeva su di esso.

Se il dato condiviso fosse stato dichiarato come *volatile*, la sincronizzazione rimarrebbe comunque necessaria. Ma verrà impedito al compilatore di ottimizzare gli accessi al dato anche dentro alla sezione critica, dove sappiamo che in realtà nessun altro può accedervi. Mentre si trattiene un lock, il dato condiviso non è *volatile*. Quando si ha a che fare con dei dati condivisi, un'opportuna sincronizzazione rende inutile l'uso di *volatile* - anzi potenzialmente dannoso.

L'uso di *volatile* fu originalmente pensato per l'accesso ai registri di I/O mappati in memoria. All'interno del kernel, l'accesso ai registri, dovrebbe essere protetto dai lock, ma si potrebbe anche desiderare che il compilatore non “ottimizzi” l'accesso ai registri all'interno di una sezione critica. Ma, all'interno del kernel, l'accesso alla memoria di I/O viene sempre fatto attraverso funzioni d'accesso; accedere alla memoria di I/O direttamente con i puntatori è sconsigliato e

non funziona su tutte le architetture. Queste funzioni d'accesso sono scritte per evitare ottimizzazioni indesiderate, quindi, di nuovo, *volatile* è inutile.

Un'altra situazione dove qualcuno potrebbe essere tentato dall'uso di *volatile*, è nel caso in cui il processore è in un'attesa attiva sul valore di una variabile. Il modo giusto di fare questo tipo di attesa è il seguente:

```
while (my_variable != what_i_want)
    cpu_relax();
```

La chiamata `cpu_relax()` può ridurre il consumo di energia del processore o cedere il passo ad un processore hyperthreaded gemello; funziona anche come una barriera per il compilatore, quindi, ancora una volta, *volatile* non è necessario. Ovviamente, tanto per puntualizzare, le attese attive sono generalmente un atto antisociale.

Ci sono comunque alcune rare situazioni dove l'uso di *volatile* nel kernel ha senso:

- Le funzioni d'accesso sopracitate potrebbero usare *volatile* su quelle architetture che supportano l'accesso diretto alla memoria di I/O. In pratica, ogni chiamata ad una funzione d'accesso diventa una piccola sezione critica a se stante, e garantisce che l'accesso avvenga secondo le aspettative del programmatore.
- I codice *inline assembly* che fa cambiamenti nella memoria, ma che non ha altri effetti esplicativi, rischia di essere rimosso da GCC. Aggiungere la parola chiave *volatile* a questo codice ne previene la rimozione.
- La variabile jiffies è speciale in quanto assume un valore diverso ogni volta che viene letta ma può essere letta senza alcuna sincronizzazione. Quindi jiffies può essere *volatile*, ma l'aggiunta ad altre variabili di questo è sconsigliata. Jiffies è considerata uno "stupido retaggio" (parole di Linus) in questo contesto; correggerla non ne varrebbe la pena e causerebbe più problemi.
- I puntatori a delle strutture dati in una memoria coerente che potrebbe essere modificata da dispositivi di I/O può, a volte, essere legittimamente *volatile*. Un esempio pratico può essere quello di un adattatore di rete che utilizza un puntatore ad un buffer circolare, questo viene cambiato dall'adattatore per indicare quali descrittori sono stati processati.

Per la maggior parte del codice, nessuna delle giustificazioni sopracitate può essere considerata. Di conseguenza, l'uso di *volatile* è probabile che venga visto come un baco e porterà a verifiche aggiuntive. Gli sviluppatori tentati dall'uso di *volatile* dovrebbero fermarsi e pensare a cosa vogliono davvero ottenere.

Le modifiche che rimuovono variabili *volatile* sono generalmente ben accette - purché accompagnate da una giustificazione che dimostri che i problemi di concorrenza siano stati opportunamente considerati.

## Riferimenti

[1] <http://lwn.net/Articles/233481/>

[2] <http://lwn.net/Articles/233482/>

## Crediti

Impulso e ricerca originale di Randy Dunlap

Scritto da Jonathan Corbet

Migliorato dai commenti di Satyam Sharma, Johannes Stezenbach, Jesper Juhl, Heikki Orsila, H. Peter Anvin, Philipp Hahn, e Stefan Richter.

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/process/clang-format.rst

**Translator** Federico Vaga <[federico.vaga@vaga.pv.it](mailto:federico.vaga@vaga.pv.it)>

## clang-format

clang-format è uno strumento per formattare codice C/C++/... secondo un gruppo di regole ed euristiche. Come tutti gli strumenti, non è perfetto e non copre tutti i singoli casi, ma è abbastanza buono per essere utile.

clang-format può essere usato per diversi fini:

- Per riformattare rapidamente un blocco di codice secondo lo stile del kernel. Particolarmente utile quando si sposta del codice e lo si allinea/ordina. Vedere [it\\_clangformatreformat](#).
- Identificare errori di stile, refusi e possibili miglioramenti nei file che mantieni, le modifiche che revisioni, le differenze, eccetera. Vedere [it\\_clangformatreview](#).
- Ti aiuta a seguire lo stile del codice, particolarmente utile per i nuovi arrivati o per coloro che lavorano allo stesso tempo su diversi progetti con stili di codifica differenti.

Il suo file di configurazione è .clang-format e si trova nella cartella principale dei sorgenti del kernel. Le regole scritte in quel file tentano di approssimare lo stile di codifica del kernel. Si tenta anche di seguire il più possibile [Documentation/translations/it\\_IT/process/coding-style.rst](#). Dato che non tutto il kernel segue lo stesso stile, potreste voler aggiustare le regole di base per un particolare sottosistema o cartella. Per farlo, potete sovrascriverle scrivendole in un altro file .clang-format in una sottocartella.

Questo strumento è già stato incluso da molto tempo nelle distribuzioni Linux più popolari. Cercate clang-format nel vostro repository. Altrimenti, potete scaricare una versione pre-generata dei binari di LLVM/clang oppure generarlo dai codici sorgenti:

<http://releases.llvm.org/download.html>

Troverete più informazioni ai seguenti indirizzi:

<https://clang.llvm.org/docs/ClangFormat.html>

<https://clang.llvm.org/docs/ClangFormatStyleOptions.html>

### Revisionare lo stile di codifica per file e modifiche

Eseguendo questo programma, potrete revisionare un intero sottosistema, cartella o singoli file alla ricerca di errori di stile, refusi o miglioramenti.

Per farlo, potete eseguire qualcosa del genere:

```
# Make sure your working directory is clean!
clang-format -i kernel/*.ch
```

E poi date un'occhiata a *git diff*.

Osservare le righe di questo diff è utile a migliorare/aggiustare le opzioni di stile nel file di configurazione; così come per verificare le nuove funzionalità/versioni di `clang-format`.

`clang-format` è in grado di leggere diversi diff unificati, quindi potrete revisionare facilmente delle modifiche e *git diff*. La documentazione si trova al seguente indirizzo:

<https://clang.llvm.org/docs/ClangFormat.html#script-for-patch-reformatting>

Per evitare che `clang-format` formatti alcune parti di un file, potete scrivere nel codice:

```
int formatted_code;
// clang-format off
void unformatted_code ;
// clang-format on
void formatted_code_again;
```

Nonostante si attraente l'idea di utilizzarlo per mantenere un file sempre in sintonia con `clang-format`, specialmente per file nuovi o se siete un manutentore, ricordatevi che altre persone potrebbero usare una versione diversa di `clang-format` oppure non utilizzarlo del tutto. Quindi, dovreste trattenervi dall'usare questi marcatori nel codice del kernel; almeno finché non vediamo che `clang-format` è diventato largamente utilizzato.

### Riformattare blocchi di codice

Utilizzando dei plugin per il vostro editor, potete riformattare una blocco (selezione) di codice con una singola combinazione di tasti. Questo è particolarmente utile: quando si riorganizza il codice, per codice complesso, macro multi-riga (e allineare le loro "barre"), eccetera.

Ricordatevi che potete sempre aggiustare le modifiche in quei casi dove questo strumento non ha fatto un buon lavoro. Ma come prima approssimazione, può essere davvero molto utile.

Questo programma si integra con molti dei più popolari editor. Alcuni di essi come vim, emacs, BBEdit, Visaul Studio, lo supportano direttamente. Al seguente indirizzo troverete le istruzioni:

<https://clang.llvm.org/docs/ClangFormat.html>

Per Atom, Eclipse, Sublime Text, Visual Studio Code, XCode e altri editor e IDEs dovreste essere in grado di trovare dei plugin pronti all'uso.

Per questo caso d'uso, considerate l'uso di un secondo `.clang-format` che potete personalizzare con le vostre opzioni. Consultare [it\\_clangformatextra](#).

## Cose non supportate

`clang-format` non ha il supporto per alcune cose che sono comuni nel codice del kernel. Sono facili da ricordare; quindi, se lo usate regolarmente, imparerete rapidamente a evitare/ignorare certi problemi.

In particolare, quelli più comuni che noterete sono:

- Allineamento di `#define` su una singola riga, per esempio:

```
#define TRACING_MAP_BITS_DEFAULT      11
#define TRACING_MAP_BITS_MAX          17
#define TRACING_MAP_BITS_MIN          7
```

contro:

```
#define TRACING_MAP_BITS_DEFAULT 11
#define TRACING_MAP_BITS_MAX    17
#define TRACING_MAP_BITS_MIN    7
```

- Allineamento dei valori iniziali, per esempio:

```
static const struct file_operations uprobe_events_ops = {
    .owner        = THIS_MODULE,
    .open         = probes_open,
    .read         = seq_read,
    .llseek       = seq_llseek,
    .release     = seq_release,
    .write        = probes_write,
};
```

contro:

```
static const struct file_operations uprobe_events_ops = {
    .owner = THIS_MODULE,
    .open = probes_open,
    .read = seq_read,
    .llseek = seq_llseek,
    .release = seq_release,
    .write = probes_write,
};
```

### Funzionalità e opzioni aggiuntive

Al fine di minimizzare le differenze fra il codice attuale e l'output del programma, alcune opzioni di stile e funzionalità non sono abilitate nella configurazione base. In altre parole, lo scopo è di rendere le differenze le più piccole possibili, permettendo la semplificazione della revisione di file, differenze e modifiche.

In altri casi (per esempio un particolare sottosistema/cartella/file), lo stile del kernel potrebbe essere diverso e abilitare alcune di queste opzioni potrebbe dare risultati migliori.

Per esempio:

- Allineare assegnamenti (`AlignConsecutiveAssignments`).
- Allineare dichiarazioni (`AlignConsecutiveDeclarations`).
- Riorganizzare il testo nei commenti (`ReflowComments`).
- Ordinare gli `#include` (`SortIncludes`).

Piuttosto che per interi file, solitamente sono utili per la riformattazione di singoli blocchi. In alternativa, potete creare un altro file `.clang-format` da utilizzare con il vostro editor/IDE.

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** *arch/riscv maintenance guidelines for developers*

**Translator** Federico Vaga <[federico.vaga@vaga.pv.it](mailto:federico.vaga@vaga.pv.it)>

## arch/riscv linee guida alla manutenzione per gli sviluppatori

### Introduzione

L'insieme di istruzioni RISC-V sono sviluppate in modo aperto: le bozze in fase di sviluppo sono disponibili a tutti per essere revisionate e per essere sperimentare nelle implementazioni. Le bozze dei nuovi moduli o estensioni possono cambiare in fase di sviluppo - a volte in modo incompatibile rispetto a bozze precedenti. Questa flessibilità può portare a dei problemi di manutenzione per il supporto RISC-V nel kernel Linux. I manutentori Linux non amano l'abbandono del codice, e il processo di sviluppo del kernel preferisce codice ben revisionato e testato rispetto a quello sperimentale. Desideriamo estendere questi stessi principi al codice relativo all'architettura RISC-V che verrà accettato per l'inclusione nel kernel.

## In aggiunta alla lista delle verifiche da fare prima di inviare una patch

Accetteremo le patch per un nuovo modulo o estensione se la fondazione RISC-V li classifica come "Frozen" o "Retified". (Ovviamente, gli sviluppatori sono liberi di mantenere una copia del kernel Linux contenente il codice per una bozza di estensione).

In aggiunta, la specifica RISC-V permette agli implementatori di creare le proprie estensioni. Queste estensioni non passano attraverso il processo di revisione della fondazione RISC-V. Per questo motivo, al fine di evitare complicazioni o problemi di prestazioni, accetteremo patch solo per quelle estensioni che sono state ufficialmente accettate dalla fondazione RISC-V. (Ovviamente, gli implementatori sono liberi di mantenere una copia del kernel Linux contenente il codice per queste specifiche estensioni).

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Note:** Per leggere la documentazione originale in inglese: Documentation/doc-guide/index.rst

## Come scrivere la documentazione del kernel

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Note:** Per leggere la documentazione originale in inglese: Documentation/doc-guide/index.rst

## Introduzione

Il kernel Linux usa [Sphinx](#) per la generazione della documentazione a partire dai file `reStructuredText` che si trovano nella cartella `Documentation`. Per generare la documentazione in HTML o PDF, usate comandi `htmldocs` o `pdfdocs`. La documentazione così generata sarà disponibile nella cartella `Documentation/output`.

I file `reStructuredText` possono contenere delle direttive che permettono di includere i commenti di documentazione, o di tipo `kernel-doc`, dai file sorgenti. Solitamente questi commenti sono utilizzati per descrivere le funzioni, i tipi e l'architettura del codice. I commenti di tipo `kernel-doc` hanno una struttura e formato speciale, ma a parte questo vengono processati come `reStructuredText`.

Inoltre, ci sono migliaia di altri documenti in formato testo sparsi nella cartella `Documentation`. Alcuni di questi verranno probabilmente convertiti, nel tempo, in formato `reStructuredText`, ma la maggior parte di questi rimarranno in formato testo.

### Installazione Sphinx

I marcatori ReST utilizzati nei file in Documentation/ sono pensati per essere processati da Sphinx nella versione 1.7 o superiore.

Esiste uno script che verifica i requisiti Sphinx. Per ulteriori dettagli consultate [Verificare le dipendenze Sphinx](#).

La maggior parte delle distribuzioni Linux forniscono Sphinx, ma l'insieme dei programmi e librerie è fragile e non è raro che dopo un aggiornamento di Sphinx, o qualche altro pacchetto Python, la documentazione non venga più generata correttamente.

Un modo per evitare questo genere di problemi è quello di utilizzare una versione diversa da quella fornita dalla vostra distribuzione. Per fare questo, vi raccomandiamo di installare Sphinx dentro ad un ambiente virtuale usando `virtualenv-3` o `virtualenv` a seconda di come Python 3 è stato pacchettizzato dalla vostra distribuzione.

---

#### Note:

- 1) Viene raccomandato l'uso del tema RTD per la documentazione in HTML. A seconda della versione di Sphinx, potrebbe essere necessaria l'installazione tramite il comando `pip install sphinx_rtd_theme`.
  - 2) Alcune pagine ReST contengono delle formule matematiche. A causa del modo in cui Sphinx funziona, queste espressioni sono scritte utilizzando LaTeX. Per una corretta interpretazione, è necessario aver installato texlive con i pacchetti amdfonts e amsmath.
- 

Riassumendo, se volete installare la versione 2.4.4 di Sphinx dovete eseguire:

```
$ virtualenv sphinx_2.4.4
$ . sphinx_2.4.4/bin/activate
(sphinx_2.4.4) $ pip install -r Documentation/sphinx/requirements.txt
```

Dopo aver eseguito `. sphinx_2.4.4/bin/activate`, il prompt cambierà per indicare che state usando il nuovo ambiente. Se aprirete una nuova sessione, prima di generare la documentazione, dovete rieseguire questo comando per rientrare nell'ambiente virtuale.

### Generazione d'immagini

Il meccanismo che genera la documentazione del kernel contiene un'estensione capace di gestire immagini in formato Graphviz e SVG (per maggior informazioni vedere [Figure ed immagini](#)).

Per far sì che questo funzioni, dovete installare entrambi i pacchetti Graphviz e ImageMagick. Il sistema di generazione della documentazione è in grado di procedere anche se questi pacchetti non sono installati, ma il risultato, ovviamente, non includerà le immagini.

## Generazione in PDF e LaTeX

Al momento, la generazione di questi documenti è supportata solo dalle versioni di Sphinx superiori alla 2.4.

Per la generazione di PDF e LaTeX, avrete bisogno anche del pacchetto XeLaTeX nella versione 3.14159265

Per alcune distribuzioni Linux potrebbe essere necessario installare anche una serie di pacchetti texlive in modo da fornire il supporto minimo per il funzionamento di XeLaTeX.

## Verificare le dipendenze Sphinx

Esiste uno script che permette di verificare automaticamente le dipendenze di Sphinx. Se lo script riesce a riconoscere la vostra distribuzione, allora sarà in grado di darvi dei suggerimenti su come procedere per completare l'installazione:

```
$ ./scripts/sphinx-pre-install
Checking if the needed tools for Fedora release 26 (Twenty Six) are available
Warning: better to also install "texlive-luatex85".
You should run:

    sudo dnf install -y texlive-luatex85
    /usr/bin/virtualenv sphinx_2.4.4
    . sphinx_2.4.4/bin/activate
    pip install -r Documentation/sphinx/requirements.txt

Can't build as 1 mandatory dependency is missing at ./scripts/sphinx-pre-
→install line 468.
```

L'impostazione predefinita prevede il controllo dei requisiti per la generazione di documenti html e PDF, includendo anche il supporto per le immagini, le espressioni matematiche e LaTeX; inoltre, presume che venga utilizzato un ambiente virtuale per Python. I requisiti per generare i documenti html sono considerati obbligatori, gli altri sono opzionali.

Questo script ha i seguenti parametri:

- no-pdf** Disabilita i controlli per la generazione di PDF;
- no-virtualenv** Utilizza l'ambiente predefinito dal sistema operativo invece che l'ambiente virtuale per Python;

## Generazione della documentazione Sphinx

Per generare la documentazione in formato HTML o PDF si eseguono i rispettivi comandi `make htmldocs` o `make pdfdocs`. Esistono anche altri formati in cui è possibile generare la documentazione; per maggiori informazioni potere eseguire il comando `make help`. La documentazione così generata sarà disponibile nella sottocartella `Documentation/output`.

Ovviamente, per generare la documentazione, Sphinx (`sphinx-build`) dev'essere installato. Se disponibile, il tema *Read the Docs* per Sphinx verrà utilizzato per ottenere una documentazione HTML più gradevole. Per la documentazione in formato PDF, invece, avrete bisogno di XeLaTeX`

e di ``convert(1) disponibile in ImageMagick (<https://www.imagemagick.org>). Tipicamente, tutti questi pacchetti sono disponibili e pacchettizzati nelle distribuzioni Linux.

Per poter passare ulteriori opzioni a Sphinx potete utilizzare la variabile make SPHINXOPTS. Per esempio, se volete che Sphinx sia più verboso durante la generazione potete usare il seguente comando make SPHINXOPTS=-v htmldocs.

Potete eliminare la documentazione generata tramite il comando make cleandocs.

## Scrivere la documentazione

Aggiungere nuova documentazione è semplice:

1. aggiungete un file .rst nella sottocartella Documentation
2. aggiungete un riferimento ad esso nell'indice (TOC tree) in Documentation/index.rst.

Questo, di solito, è sufficiente per la documentazione più semplice (come quella che state leggendo ora), ma per una documentazione più elaborata è consigliato creare una sottocartella dedicata (o, quando possibile, utilizzarne una già esistente). Per esempio, il sottosistema grafico è documentato nella sottocartella Documentation/gpu; questa documentazione è divisa in diversi file .rst ed un indice index.rst (con un toctree dedicato) a cui si fa riferimento nell'indice principale.

Consultate la documentazione di [Sphinx](#) e [reStructuredText](#) per maggiori informazioni circa le loro potenzialità. In particolare, il [manuale introduttivo a reStructuredText](#) di Sphinx è un buon punto da cui cominciare. Esistono, inoltre, anche alcuni [costruttori specifici per Sphinx](#).

## Guide linea per la documentazione del kernel

In questa sezione troverete alcune linee guida specifiche per la documentazione del kernel:

- Non esagerate con i costrutti di reStructuredText. Mantenete la documentazione semplice. La maggior parte della documentazione dovrebbe essere testo semplice con una strutturazione minima che permetta la conversione in diversi formati.
- Mantenete la strutturazione il più fedele possibile all'originale quando convertite un documento in formato reStructuredText.
- Aggiornate i contenuti quando convertite della documentazione, non limitatevi solo alla formattazione.
- Mantenete la decorazione dei livelli di intestazione come segue:

1. = con una linea superiore per il titolo del documento:

```
=====
Titolo
=====
```

2. = per i capitoli:

```
Capitoli
=====
```

3. - per le sezioni:

```
Sezioni
-----

```

4. ~ per le sottosezioni:

```
Sottosezioni
~~~~~
```

Sebbene RST non forzi alcun ordine specifico (*Piuttosto che imporre un numero ed un ordine fisso di decorazioni, l'ordine utilizzato sarà quello incontrato*), avere uniformità dei livelli principali rende più semplice la lettura dei documenti.

- Per inserire blocchi di testo con caratteri a dimensione fissa (codici di esempio, casi d'uso, eccetera): utilizzate :: quando non è necessario evidenziare la sintassi, specialmente per piccoli frammenti; invece, utilizzate .. code-block:: <language> per blocchi più lunghi che beneficeranno della sintassi evidenziata. Per un breve pezzo di codice da inserire nel testo, usate ``.

## Il dominio C

Il **Dominio Sphinx C** (denominato c) è adatto alla documentazione delle API C. Per esempio, un prototipo di una funzione:

```
.. c:function:: int ioctl( int fd, int request )
```

Il dominio C per kernel-doc ha delle funzionalità aggiuntive. Per esempio, potete assegnare un nuovo nome di riferimento ad una funzione con un nome molto comune come open o ioctl:

```
.. c:function:: int ioctl( int fd, int request )
:name: VIDIOC_LOG_STATUS
```

Il nome della funzione (per esempio ioctl) rimane nel testo ma il nome del suo riferimento cambia da ioctl a VIDIOC\_LOG\_STATUS. Anche la voce nell'indice cambia in VIDIOC\_LOG\_STATUS.

Notate che per una funzione non c'è bisogno di usare c:func: per generarne i riferimenti nella documentazione. Grazie a qualche magica estensione a Sphinx, il sistema di generazione della documentazione trasformerà automaticamente un riferimento ad una funzione() in un riferimento incrociato quando questa ha una voce nell'indice. Se trovate degli usi di c:func: nella documentazione del kernel, sentitevi liberi di rimuoverli.

## Tabelle a liste

Raccomandiamo l'uso delle tabelle in formato lista (*list table*). Le tabelle in formato lista sono liste di liste. In confronto all'ASCII-art potrebbero non apparire di facile lettura nei file in formato testo. Il loro vantaggio è che sono facili da creare o modificare e che la differenza di una modifica è molto più significativa perché limitata alle modifiche del contenuto.

La flat-table è anch'essa una lista di liste simile alle list-table ma con delle funzionalità aggiuntive:

- column-span: col ruolo `cspan` una cella può essere estesa attraverso colonne successive
- raw-span: col ruolo `rspan` una cella può essere estesa attraverso righe successive
- auto-span: la cella più a destra viene estesa verso destra per compensare la mancanza di celle. Con l'opzione `:fill-cells`: questo comportamento può essere cambiato da `auto-span` ad `auto-fill`, il quale inserisce automaticamente celle (vuote) invece che estendere l'ultima.

opzioni:

- `:header-rows`: [int] conta le righe di intestazione
- `:stub-columns`: [int] conta le colonne di stub
- `:widths`: [[int] [int] ... ] larghezza delle colonne
- `:fill-cells`: invece di estendere automaticamente una cella su quelle mancanti, ne crea di vuote.

ruoli:

- `:cspan`: [int] colonne successive (*morecols*)
- `:rspan`: [int] righe successive (*morerows*)

L'esempio successivo mostra come usare questo marcatore. Il primo livello della nostra lista di liste è la *riga*. In una *riga* è possibile inserire solamente la lista di celle che compongono la *riga* stessa. Fanno eccezione i *commenti* ( . . ) ed i *collegamenti* (per esempio, un riferimento a `:ref:`last row <last row>` / last row`)

```
.. flat-table:: table title
  :widths: 2 1 1 3

  * - head col 1
    - head col 2
    - head col 3
    - head col 4

  * - row 1
    - field 1.1
    - field 1.2 with autospan

  * - row 2
    - field 2.1
    - :rspan:`1` :cspan:`1` field 2.2 - 3.3

  * .. _`it last row`:
    - row 3
```

Che verrà rappresentata nel seguente modo:

Table 2: table title

head col 1	head col 2	head col 3	head col 4
row 1	field 1.1	field 1.2 with colspan	
row 2	field 2.1	field 2.2 - 3.3	
row 3			

## Riferimenti incrociati

Aggiungere un riferimento incrociato da una pagina della documentazione ad un'altra può essere fatto scrivendo il percorso al file corrispondente, non serve alcuna sintassi speciale. Si possono usare sia percorsi assoluti che relativi. Quelli assoluti iniziano con "documentation/". Per esempio, potete fare riferimento a questo documento in uno dei seguenti modi (da notare che l'estensione .rst è necessaria):

Vedere Documentation/doc-guide/sphinx.rst. Questo funziona sempre  
Guardate pshinx.rst, che si trova nella stessa cartella.  
Leggete ../sphinx.rst, che si trova nella cartella precedente.

Se volete che il collegamento abbia un testo diverso rispetto al titolo del documento, allora dovrete usare la direttiva Sphinx doc. Per esempio:

Vedere :doc:`il mio testo per il collegamento <sphinx>`.

Nella maggioranza dei casi si consiglia il primo metodo perché è più pulito ed adatto a chi legge dai sorgenti. Se incontrare un :doc: che non da alcun valore, sentitevi liberi di convertirlo in un percorso al documento.

Per informazioni riguardo ai riferimenti incrociati ai commenti kernel-doc per funzioni o tipi, consultate

## Figure ed immagini

Se volete aggiungere un'immagine, utilizzate le direttive `kernel-figure` e `kernel-image`. Per esempio, per inserire una figura di un'immagine in formato SVG (*Una semplice immagine SVG*):

```
.. kernel-figure:: ../../doc-guide/svg_image.svg
   :alt: una semplice immagine SVG

   Una semplice immagine SVG
```

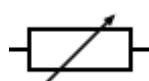


Fig. 1: Una semplice immagine SVG

Le direttive del kernel per figure ed immagini supportano il formato **DOT**, per maggiori informazioni

- DOT: <http://graphviz.org/pdf/dotguide.pdf>

- Graphviz: <http://www.graphviz.org/content/dot-language>

Un piccolo esempio (*Esempio DOT*):

```
.. kernel-figure:: ../../../../doc-guide/hello.dot
:alt: ciao mondo
```

Esempio DOT

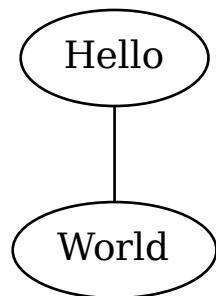


Fig. 2: Esempio DOT

Tramite la direttiva `kernel-render` è possibile aggiungere codice specifico; ad esempio nel formato **DOT** di Graphviz.:

```
.. kernel-render:: DOT
:alt: foobar digraph
:caption: Codice **DOT** (Graphviz) integrato

digraph foo {
    "bar" -> "baz";
}
```

La rappresentazione dipenderà dei programmi installati. Se avete Graphviz installato, vedrete un'immagine vettoriale. In caso contrario, il codice grezzo verrà rappresentato come *blocco testuale* (*Codice DOT (Graphviz) integrato*).

La direttiva `render` ha tutte le opzioni della direttiva `figure`, con l'aggiunta dell'opzione `caption`. Se `caption` ha un valore allora un nodo `figure` viene aggiunto. Altrimenti verrà aggiunto un nodo `image`. L'opzione `caption` è necessaria in caso si vogliano aggiungere dei riferimenti (*Integrare codice SVG*).

Per la scrittura di codice **SVG**:

```
.. kernel-render:: SVG
:caption: Integrare codice **SVG**
:alt: so-nw-arrow

<?xml version="1.0" encoding="UTF-8"?>
```

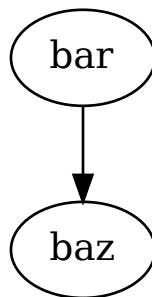


Fig. 3: Codice **DOT** (Graphviz) integrato

```

<svg xmlns="http://www.w3.org/2000/svg" version="1.1" ...>
  ...
</svg>
  
```



Fig. 4: Integrare codice **SVG**

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Note:** Per leggere la documentazione originale in inglese: Documentation/doc-guide/index.rst

### Scrivere i commenti in kernel-doc

Nei file sorgenti del kernel Linux potrete trovare commenti di documentazione strutturanti secondo il formato kernel-doc. Essi possono descrivere funzioni, tipi di dati, e l'architettura del codice.

**Note:** Il formato kernel-doc può sembrare simile a gtk-doc o Doxygen ma in realtà è molto differente per ragioni storiche. I sorgenti del kernel contengono decine di migliaia di commenti kernel-doc. Siete pregati d'attenervi allo stile qui descritto.

La struttura kernel-doc è estratta a partire dai commenti; da questi viene generato il dominio Sphinx per il C con un'adeguata descrizione per le funzioni ed i tipi di dato con i loro relativi collegamenti. Le descrizioni vengono filtrate per cercare i riferimenti ed i marcatori.

Vedere di seguito per maggiori dettagli.

Tutte le funzioni esportate verso i moduli esterni utilizzando `EXPORT_SYMBOL` o `EXPORT_SYMBOL_GPL` dovrebbero avere un commento kernel-doc. Quando l'intenzione è di utilizzarle nei moduli, anche le funzioni e le strutture dati nei file d'intestazione dovrebbero avere dei commenti kernel-doc.

È considerata una buona pratica quella di fornire una documentazione formattata secondo kernel-doc per le funzioni che sono visibili da altri file del kernel (ovvero, che non siano dichiarate utilizzando `static`). Raccomandiamo, inoltre, di fornire una documentazione kernel-doc anche per procedure private (ovvero, dichiarate "static") al fine di fornire una struttura più coerente dei sorgenti. Quest'ultima raccomandazione ha una priorità più bassa ed è a discrezione dal manutentore (MAINTAINER) del file sorgente.

Sicuramente la documentazione formattata con kernel-doc è necessaria per le funzioni che sono esportate verso i moduli esterni utilizzando `EXPORT_SYMBOL` o `EXPORT_SYMBOL_GPL`.

Cerchiamo anche di fornire una documentazione formattata secondo kernel-doc per le funzioni che sono visibili da altri file del kernel (ovvero, che non siano dichiarate utilizzando "static")

Raccomandiamo, inoltre, di fornire una documentazione formattata con kernel-doc anche per procedure private (ovvero, dichiarate "static") al fine di fornire una struttura più coerente dei sorgenti. Questa raccomandazione ha una priorità più bassa ed è a discrezione dal manutentore (MAINTAINER) del file sorgente.

Le strutture dati visibili nei file di intestazione dovrebbero essere anch'esse documentate utilizzando commenti formattati con kernel-doc.

### Come formattare i commenti kernel-doc

I commenti kernel-doc iniziano con il marcitore `/**`. Il programma `kernel-doc` estrarrà i commenti marchiati in questo modo. Il resto del commento è formattato come un normale commento multilinea, ovvero con un asterisco all'inizio d'ogni riga e che si conclude con `*/` su una riga separata.

I commenti kernel-doc di funzioni e tipi dovrebbero essere posizionati appena sopra la funzione od il tipo che descrivono. Questo allo scopo di aumentare la probabilità che chi cambia il codice si ricordi di aggiornare anche la documentazione. I commenti kernel-doc di tipo più generale possono essere posizionati ovunque nel file.

Al fine di verificare che i commenti siano formattati correttamente, potete eseguire il programma `kernel-doc` con un livello di verbosità alto e senza che questo produca alcuna documentazione. Per esempio:

```
scripts/kernel-doc -v -none drivers/foo/bar.c
```

Il formato della documentazione è verificato dalla procedura di generazione del kernel quando viene richiesto di effettuare dei controlli extra con GCC:

```
make W=n
```

## Documentare le funzioni

Generalmente il formato di un commento kernel-doc per funzioni e macro simil-funzioni è il seguente:

```
/**  
 * function_name() - Brief description of function.  
 * @arg1: Describe the first argument.  
 * @arg2: Describe the second argument.  
 *         One can provide multiple line descriptions  
 *         for arguments.  
 *  
 * A longer description, with more discussion of the function function_name()  
 * that might be useful to those using or modifying it. Begins with an  
 * empty comment line, and may include additional embedded empty  
 * comment lines.  
 *  
 * The longer description may have multiple paragraphs.  
 *  
 * Context: Describes whether the function can sleep, what locks it takes,  
 *         releases, or expects to be held. It can extend over multiple  
 *         lines.  
 * Return: Describe the return value of function_name.  
 *  
 * The return value description can also have multiple paragraphs, and should  
 * be placed at the end of the comment block.  
 */
```

La descrizione introduttiva (*brief description*) che segue il nome della funzione può continuare su righe successive e termina con la descrizione di un argomento, una linea di commento vuota, oppure la fine del commento.

## Parametri delle funzioni

Ogni argomento di una funzione dovrebbe essere descritto in ordine, subito dopo la descrizione introduttiva. Non lasciare righe vuote né fra la descrizione introduttiva e quella degli argomenti, né fra gli argomenti.

Ogni @argument: può estendersi su più righe.

---

**Note:** Se la descrizione di @argument: si estende su più righe, la continuazione dovrebbe iniziare alla stessa colonna della riga precedente:

```
* @argument: some long description  
*             that continues on next lines
```

or:

```
* @argument:  
*             some long description  
*             that continues on next lines
```

Se una funzione ha un numero variabile di argomento, la sua descrizione dovrebbe essere scritta con la notazione kernel-doc:

```
* @...: description
```

### Contesto delle funzioni

Il contesto in cui le funzioni vengono chiamate viene descritto in una sezione chiamata Context. Questo dovrebbe informare sulla possibilità che una funzione dorma (*sleep*) o che possa essere chiamata in un contesto d'interruzione, così come i *lock* che prende, rilascia e che si aspetta che vengano presi dal chiamante.

Esempi:

```
* Context: Any context.  
* Context: Any context. Takes and releases the RCU lock.  
* Context: Any context. Expects <lock> to be held by caller.  
* Context: Process context. May sleep if @gfp flags permit.  
* Context: Process context. Takes and releases <mutex>.  
* Context: Softirq or process context. Takes and releases <lock>, BH-safe.  
* Context: Interrupt context.
```

### Valore di ritorno

Il valore di ritorno, se c'è, viene descritto in una sezione dedicata di nome Return.

---

#### Note:

- 1) La descrizione multiriga non riconosce il termine d'una riga, per cui se provate a formattare bene il vostro testo come nel seguente esempio:

```
* Return:  
* 0 - OK  
* -EINVAL - invalid argument  
* -ENOMEM - out of memory
```

le righe verranno unite e il risultato sarà:

```
Return: 0 - OK -EINVAL - invalid argument -ENOMEM - out of memory
```

Quindi, se volete che le righe vengano effettivamente generate, dovete utilizzare una lista ReST, ad esempio:

```
* Return:  
* * 0           - OK to runtime suspend the device  
* * -EBUSY      - Device should not be runtime suspended
```

- 2) Se il vostro testo ha delle righe che iniziano con una frase seguita dai due punti, allora ogni una di queste frasi verrà considerata come il nome di una nuova sezione, e probabilmente non produrrà gli effetti desiderati.

## Documentare strutture, unioni ed enumerazioni

Generalmente il formato di un commento kernel-doc per struct, union ed enum è:

```
/**  
 * struct struct_name - Brief description.  
 * @member1: Description of member1.  
 * @member2: Description of member2.  
 *          One can provide multiple line descriptions  
 *          for members.  
 *  
 * Description of the structure.  
 */
```

Nell'esempio qui sopra, potete sostituire `struct` con `union` o `enum` per descrivere unioni ed enumerati. `member` viene usato per indicare i membri di strutture ed unioni, ma anche i valori di un tipo enumerato.

La descrizione introduttiva (*brief description*) che segue il nome della funzione può continuare su righe successive e termina con la descrizione di un argomento, una linea di commento vuota, oppure la fine del commento.

## Membri

I membri di strutture, unioni ed enumerati devono essere documentati come i parametri delle funzioni; seguono la descrizione introduttiva e possono estendersi su più righe.

All'interno d'una struttura o d'un unione, potete utilizzare le etichette `private:` e `public:`. I campi che sono nell'area `private:` non verranno inclusi nella documentazione finale.

Le etichette `private:` e `public:` devono essere messe subito dopo il marcitore di un commento `/*`. Opzionalmente, possono includere commenti fra `:` e il marcitore di fine commento `*/`.

Esempio:

```
/**  
 * struct my_struct - short description  
 * @a: first member  
 * @b: second member  
 * @d: fourth member  
 *  
 * Longer description  
 */  
struct my_struct {  
    int a;  
    int b;  
/* private: internal use only */
```

```
    int c;
/* public: the next one is public */
    int d;
};
```

### Strutture ed unioni annidate

È possibile documentare strutture ed unioni annidate, ad esempio:

```
/** 
 * struct nested_foobar - a struct with nested unions and structs
 * @memb1: first member of anonymous union/anonymous struct
 * @memb2: second member of anonymous union/anonymous struct
 * @memb3: third member of anonymous union/anonymous struct
 * @memb4: fourth member of anonymous union/anonymous struct
 * @bar: non-anonymous union
 * @bar.st1: struct st1 inside @bar
 * @bar.st2: struct st2 inside @bar
 * @bar.st1.memb1: first member of struct st1 on union bar
 * @bar.st1.memb2: second member of struct st1 on union bar
 * @bar.st2.memb1: first member of struct st2 on union bar
 * @bar.st2.memb2: second member of struct st2 on union bar
 */
struct nested_foobar {
    /* Anonymous union/struct*/
    union {
        struct {
            int memb1;
            int memb2;
        }
        struct {
            void *memb3;
            int memb4;
        }
    }
    union {
        struct {
            int memb1;
            int memb2;
        } st1;
        struct {
            void *memb1;
            int memb2;
        } st2;
    } bar;
};
```

---

**Note:**

- 1) Quando documentate una struttura od unione annidata, ad esempio di nome foo, il suo campo bar dev'essere documentato usando @foo.bar:
- 2) Quando la struttura od unione annidata è anonima, il suo campo bar dev'essere documentato usando @bar:

## Commenti in linea per la documentazione dei membri

I membri d'una struttura possono essere documentati in linea all'interno della definizione stessa. Ci sono due stili: una singola riga di commento che inizia con `/**` e finisce con `*/`; commenti multi riga come qualsiasi altro commento kernel-doc:

```
/**
 * struct foo - Brief description.
 * @foo: The Foo member.
 */
struct foo {
    int foo;
    /**
     * @bar: The Bar member.
     */
    int bar;
    /**
     * @baz: The Baz member.
     *
     * Here, the member description may contain several paragraphs.
     */
    int baz;
    union {
        /** @foobar: Single line description. */
        int foobar;
    };
    /** @bar2: Description for struct @bar2 inside @foo */
    struct {
        /**
         * @bar2.barbar: Description for @barbar inside @foo.bar2
         */
        int barbar;
    } bar2;
};
```

### Documentazione dei tipi di dato

Generalmente il formato di un commento kernel-doc per typedef è il seguente:

```
/**  
 * typedef type_name - Brief description.  
 *  
 * Description of the type.  
 */
```

Anche i tipi di dato per prototipi di funzione possono essere documentati:

```
/**  
 * typedef type_name - Brief description.  
 * @arg1: description of arg1  
 * @arg2: description of arg2  
 *  
 * Description of the type.  
 *  
 * Context: Locking context.  
 * Return: Meaning of the return value.  
 */  
typedef void (*type_name)(struct v4l2_ctrl *arg1, void *arg2);
```

### Marcatori e riferimenti

All'interno dei commenti di tipo kernel-doc vengono riconosciuti i seguenti *pattern* che vengono convertiti in marcatori reStructuredText ed in riferimenti del dominio Sphinx per il C.

**Attention:** Questi sono riconosciuti **solo** all'interno di commenti kernel-doc, e **non** all'interno di documenti reStructuredText.

**funcname()** Riferimento ad una funzione.

**@parameter** Nome di un parametro di una funzione (nessun riferimento, solo formattazione).

**%CONST** Il nome di una costante (nessun riferimento, solo formattazione)

**``literal``** Un blocco di testo che deve essere riportato così com'è. La rappresentazione finale utilizzerà caratteri a spaziatura fissa.

Questo è utile se dovete utilizzare caratteri speciali che altrimenti potrebbero assumere un significato diverso in kernel-doc o in reStructuredText

Questo è particolarmente utile se dovete scrivere qualcosa come %ph all'interno della descrizione di una funzione.

**\$ENVVAR** Il nome di una variabile d'ambiente (nessun riferimento, solo formattazione).

**&struct name** Riferimento ad una struttura.

**&enum name** Riferimento ad un'enumerazione.

**&typedef name** Riferimento ad un tipo di dato.

**&struct\_name->member or &struct\_name.member** Riferimento ad un membro di una struttura o di un'unione. Il riferimento sarà la struttura o l'unione, non il membro.

**&name** Un generico riferimento ad un tipo. Usate, preferibilmente, il riferimento completo come descritto sopra. Questo è dedicato ai commenti obsoleti.

## Riferimenti usando reStructuredText

Nei documenti reStructuredText non serve alcuna sintassi speciale per fare riferimento a funzioni e tipi definiti nei commenti kernel-doc. Sarà sufficiente terminare i nomi di funzione con (), e scrivere struct, union, enum, o typedef prima di un tipo. Per esempio:

```
See foo()
See struct foo.
See union bar.
See enum baz.
See typedef meh.
```

Tuttavia, la personalizzazione dei collegamenti è possibile solo con la seguente sintassi:

```
See :c:func:`my custom link text for function foo <foo>`.
See :c:type:`my custom link text for struct bar <bar>`.
```

## Commenti per una documentazione generale

Al fine d'avere il codice ed i commenti nello stesso file, potete includere dei blocchi di documentazione kernel-doc con un formato libero invece che nel formato specifico per funzioni, strutture, unioni, enumerati o tipi di dato. Per esempio, questo tipo di commento potrebbe essere usato per la spiegazione delle operazioni di un driver o di una libreria

Questo s'ottiene utilizzando la parola chiave DOC: a cui viene associato un titolo.

Generalmente il formato di un commento generico o di visione d'insieme è il seguente:

```
/** 
 * DOC: Theory of Operation
 *
 * The whizbang foobar is a dilly of a gizmo. It can do whatever you
 * want it to do, at any time. It reads your mind. Here's how it works.
 *
 * foo bar splat
 *
 * The only drawback to this gizmo is that it can sometimes damage
 * hardware, software, or its subject(s).
 */
```

Il titolo che segue DOC: funziona da intestazione all'interno del file sorgente, ma anche come identificatore per l'estrazione di questi commenti di documentazione. Quindi, il titolo dev'essere unico all'interno del file.

### Includere i commenti di tipo kernel-doc

I commenti di documentazione possono essere inclusi in un qualsiasi documento di tipo reStructuredText mediante l'apposita direttiva nell'estensione kernel-doc per Sphinx.

Le direttive kernel-doc sono nel formato:

```
.. kernel-doc:: source  
   :option:
```

Il campo *source* è il percorso ad un file sorgente, relativo alla cartella principale dei sorgenti del kernel. La direttiva supporta le seguenti opzioni:

**export: [*source-pattern* ...]** Include la documentazione per tutte le funzioni presenti nel file sorgente (*source*) che sono state esportate utilizzando EXPORT\_SYMBOL o EXPORT\_SYMBOL\_GPL in *source* o in qualsiasi altro *source-pattern* specificato.

Il campo *source-pattern* è utile quando i commenti kernel-doc sono stati scritti nei file d'intestazione, mentre EXPORT\_SYMBOL e EXPORT\_SYMBOL\_GPL si trovano vicino alla definizione delle funzioni.

Esempi:

```
.. kernel-doc:: lib(bitmap.c  
   :export:  
  
.. kernel-doc:: include/net/mac80211.h  
   :export: net/mac80211/*.c
```

**internal: [*source-pattern* ...]** Include la documentazione per tutte le funzioni ed i tipi presenti nel file sorgente (*source*) che **non** sono stati esportati utilizzando EXPORT\_SYMBOL o EXPORT\_SYMBOL\_GPL né in *source* né in qualsiasi altro *source-pattern* specificato.

Esempio:

```
.. kernel-doc:: drivers/gpu/drm/i915/intel_audio.c  
   :internal:
```

**identifiers: [*function/type* ...]** Include la documentazione per ogni *function* e *type* in *source*.

Se non vengono esplicitamente specificate le funzioni da includere, allora verranno incluse tutte quelle disponibili in *source*.

Esempi:

```
.. kernel-doc:: lib(bitmap.c  
   :identifiers: bitmap_parselist bitmap_parselist_user  
  
.. kernel-doc:: lib(idr.c  
   :identifiers:
```

**functions: [*function* ...]** Questo è uno pseudonimo, deprecato, per la direttiva 'identifiers'.

**doc: *title*** Include la documentazione del paragrafo DOC: identificato dal titolo (*title*) all'interno del file sorgente (*source*). Gli spazi in *title* sono permessi; non virgolettate *title*. Il campo

*title* è utilizzato per identificare un paragrafo e per questo non viene incluso nella documentazione finale. Verificate d'avere l'intestazione appropriata nei documenti reStructuredText.

Esempio:

```
.. kernel-doc:: drivers/gpu/drm/i915/intel_audio.c
:doc: High Definition Audio over HDMI and Display Port
```

Senza alcuna opzione, la direttiva kernel-doc include tutti i commenti di documentazione presenti nel file sorgente (*source*).

L'estensione kernel-doc fa parte dei sorgenti del kernel, la si può trovare in Documentation/sphinx/kerneldoc.py. Internamente, viene utilizzato lo script scripts/kernel-doc per estrarre i commenti di documentazione dai file sorgenti.

## Come utilizzare kernel-doc per generare pagine man

Se volete utilizzare kernel-doc solo per generare delle pagine man, potete farlo direttamente dai sorgenti del kernel:

```
$ scripts/kernel-doc -man $(git grep -l '/\*\*/' -- :^Documentation :^tools) |_
  ↳scripts/split-man.pl /tmp/man
```

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Note:** Per leggere la documentazione originale in inglese: Documentation/doc-guide/index.rst

## Includere gli i file di intestazione uAPI

Qualche volta è utile includere dei file di intestazione e degli esempi di codice C al fine di descrivere l'API per lo spazio utente e per generare dei riferimenti fra il codice e la documentazione. Aggiungere i riferimenti ai file dell'API dello spazio utente ha ulteriori vantaggi: Sphinx genererà dei messaggi d'avviso se un simbolo non viene trovato nella documentazione. Questo permette di mantenere allineate la documentazione della uAPI (API spazio utente) con le modifiche del kernel. Il programma [parse\\_headers.pl](#) genera questi riferimenti. Esso dev'essere invocato attraverso un Makefile, mentre si genera la documentazione. Per avere un esempio su come utilizzarlo all'interno del kernel consultate Documentation/userspace-api/media/Makefile.

### parse\_headers.pl

#### NOME

parse\_headers.pl - analizza i file C al fine di identificare funzioni, strutture, enumerati e definizioni, e creare riferimenti per Sphinx

#### SINTASSI

**parse\_headers.pl** [<options>] <C\_FILE> <OUT\_FILE> [<EXCEPTIONS\_FILE>]

Dove <options> può essere: -debug, -usage o -help.

#### OPZIONI

##### **-debug**

Lo script viene messo in modalità verbosa, utile per il debugging.

##### **-usage**

Mostra un messaggio d'aiuto breve e termina.

##### **-help**

Mostra un messaggio d'aiuto dettagliato e termina.

#### DESCRIZIONE

Converte un file d'intestazione o un file sorgente C (C\_FILE) in un testo ReStructuredText incluso mediante il blocco ..parsed-literal con riferimenti alla documentazione che descrive l'API. Opzionalmente, il programma accetta anche un altro file (EXCEPTIONS\_FILE) che descrive quali elementi debbano essere ignorati o il cui riferimento deve puntare ad elemento diverso dal predefinito.

Il file generato sarà disponibile in (OUT\_FILE).

Il programma è capace di identificare *define*, funzioni, strutture, tipi di dato, enumerati e valori di enumerati, e di creare i riferimenti per ognuno di loro. Inoltre, esso è capace di distinguere le #define utilizzate per specificare i comandi ioctl di Linux.

Il file EXCEPTIONS\_FILE contiene due tipi di dichiarazioni: **ignore** o **replace**.

La sintassi per ignore è:

**ignore tipo nome**

La dichiarazione **ignore** significa che non verrà generato alcun riferimento per il simbolo **name** di tipo **tipo**.

La sintassi per replace è:

**replace tipo nome nuovo\_valore**

La dichiarazione **replace** significa che verrà generato un riferimento per il simbolo **named** di tipo **tipo**, ma, invece di utilizzare il valore predefinito, verrà utilizzato il valore **nuovo\_valore**.

Per entrambe le dichiarazioni, il **tipo** può essere uno dei seguenti:

### ioctl

La dichiarazione ignore o replace verrà applicata su definizioni di ioctl come la seguente:

```
#define VIDIOC_DBG_S_REGISTER _IOW('V', 79, struct v4l2_dbg_register)
```

### define

La dichiarazione ignore o replace verrà applicata su una qualsiasi #define trovata in C\_FILE.

### typedef

La dichiarazione ignore o replace verrà applicata ad una dichiarazione typedef in C\_FILE.

### struct

La dichiarazione ignore o replace verrà applicata ai nomi di strutture in C\_FILE.

### enum

La dichiarazione ignore o replace verrà applicata ai nomi di enumerati in C\_FILE.

### symbol

La dichiarazione ignore o replace verrà applicata ai nomi di valori di enumerati in C\_FILE.

Per le dichiarazioni di tipo replace, il campo **new\_value** utilizzerà automaticamente i riferimenti :c:type: per **typedef**, **enum** e **struct**. Invece, utilizzerà :ref: per **ioctl**, **define** e **symbol**. Il tipo di riferimento può essere definito esplicitamente nella dichiarazione stessa.

## ESEMPI

```
ignore define _VIDEODEV2_H
```

Ignora una definizione #define \_VIDEODEV2\_H nel file C\_FILE.

```
ignore symbol PRIVATE
```

In un enumerato come il seguente:

```
enum foo { BAR1, BAR2, PRIVATE };
```

Non genererà alcun riferimento per **PRIVATE**.

```
replace symbol BAR1 :c:type:`foo` replace symbol BAR2 :c:type:`foo`
```

In un enumerato come il seguente:

```
enum foo { BAR1, BAR2, PRIVATE };
```

Genererà un riferimento ai valori BAR1 e BAR2 dal simbolo foo nel dominio C.

### BUGS

Riferire ogni malfunzionamento a Mauro Carvalho Chehab <[mchehab@s-opensource.com](mailto:mchehab@s-opensource.com)>

### COPYRIGHT

Copyright (c) 2016 by Mauro Carvalho Chehab <[mchehab@s-opensource.com](mailto:mchehab@s-opensource.com)>.

Licenza GPLv2: GNU GPL version 2 <<http://gnu.org/licenses/gpl.html>>.

Questo è software libero: siete liberi di cambiarlo e ridistribuirlo. Non c'è alcuna garanzia, nei limiti permessi dalla legge.

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/kernel-hacking/index.rst

**Translator** Federico Vaga <[federico.vaga@vaga.pv.it](mailto:federico.vaga@vaga.pv.it)>

### Guida all'hacking del kernel

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

---

**Note:** Per leggere la documentazione originale in inglese: Documentation/kernel-hacking/hacking.rst

---

**Original** Documentation/kernel-hacking/hacking.rst

**Translator** Federico Vaga <[federico.vaga@vaga.pv.it](mailto:federico.vaga@vaga.pv.it)>

### L'inaffidabile guida all'hacking del kernel Linux

**Author** Rusty Russell

## Introduzione

Benvenuto, gentile lettore, alla notevole ed inaffidabile guida all'hacking del kernel Linux ad opera di Rusty. Questo documento descrive le procedure più usate ed i concetti necessari per scrivere codice per il kernel: lo scopo è di fornire ai programmati C più esperti un manuale di base per sviluppo. Eviterò dettagli implementativi: per questo abbiamo il codice, ed ignorerò intere parti di alcune procedure.

Prima di leggere questa guida, sappiate che non ho mai voluto scriverla, essendo esageratamente sotto qualificato, ma ho sempre voluto leggere qualcosa di simile, e quindi questa era l'unica via. Spero che possa crescere e diventare un compendio di buone pratiche, punti di partenza e generiche informazioni.

## Gli attori

In qualsiasi momento ognuna delle CPU di un sistema può essere:

- non associata ad alcun processo, servendo un'interruzione hardware;
- non associata ad alcun processo, servendo un softirq o tasklet;
- in esecuzione nello spazio kernel, associata ad un processo (contesto utente);
- in esecuzione di un processo nello spazio utente;

Esiste un ordine fra questi casi. Gli ultimi due possono avvicendarsi (preempt) l'un l'altro, ma a parte questo esiste una gerarchia rigida: ognuno di questi può avvicendarsi solo ad uno di quelli sottostanti. Per esempio, mentre un softirq è in esecuzione su d'una CPU, nessun altro softirq può avvicendarsi nell'esecuzione, ma un'interruzione hardware può. Ciò nonostante, le altre CPU del sistema operano indipendentemente.

Più avanti vedremo alcuni modi in cui dal contesto utente è possibile bloccare le interruzioni, così da impedirne davvero il diritto di prelazione.

## Contesto utente

Ci si trova nel contesto utente quando si arriva da una chiamata di sistema od altre eccezioni: come nello spazio utente, altre procedure più importanti, o le interruzioni, possono far valere il proprio diritto di prelazione sul vostro processo. Potete sospendere l'esecuzione chiamando `schedule()`.

---

**Note:** Si è sempre in contesto utente quando un modulo viene caricato o rimosso, e durante le operazioni nello strato dei dispositivi a blocchi (*block layer*).

---

Nel contesto utente, il puntatore `current` (il quale indica il processo al momento in esecuzione) è valido, e `in_interrupt()` (`include/linux/preempt.h`) è falsa.

**Warning:** Attenzione che se avete la prelazione o i softirq disabilitati (vedere di seguito), `in_interrupt()` ritornerà un falso positivo.

### Interruzioni hardware (Hard IRQs)

Temporizzatori, schede di rete e tastiere sono esempi di vero hardware che possono produrre interruzioni in un qualsiasi momento. Il kernel esegue i gestori d'interruzione che prestano un servizio all'hardware. Il kernel garantisce che questi gestori non vengano mai interrotti: se una stessa interruzione arriva, questa verrà accodata (o scartata). Dato che durante la loro esecuzione le interruzioni vengono disabilitate, i gestori d'interruzione devono essere veloci: spesso si limitano esclusivamente a notificare la presa in carico dell'interruzione, programmare una 'interruzione software' per l'esecuzione e quindi terminare.

Potete dire d'essere in una interruzione hardware perché `in_hardirq()` ritorna vero.

**Warning:** Attenzione, questa ritornerà un falso positivo se le interruzioni sono disabilitate (vedere di seguito).

### Contesto d'interruzione software: softirq e tasklet

Quando una chiamata di sistema sta per tornare allo spazio utente, oppure un gestore d'interruzioni termina, qualsiasi 'interruzione software' marcata come pendente (solitamente da un'interruzione hardware) viene eseguita (`kernel/softirq.c`).

La maggior parte del lavoro utile alla gestione di un'interruzione avviene qui. All'inizio della transizione ai sistemi multiprocessore, c'erano solo i cosiddetti 'bottom half' (BH), i quali non traevano alcun vantaggio da questi sistemi. Non appena abbandonammo i computer raffazionati con fiammiferi e cicche, abbandonammo anche questa limitazione e migrammo alle interruzioni software 'softirqs'.

Il file `include/linux/interrupt.h` elenca i differenti tipi di 'softirq'. Un tipo di softirq molto importante è il timer (`include/linux/timer.h`): potete programmarlo per far sì che esegua funzioni dopo un determinato periodo di tempo.

Dato che i softirq possono essere eseguiti simultaneamente su più di un processore, spesso diventa estenuante l'averci a che fare. Per questa ragione, i tasklet (`include/linux/interrupt.h`) vengo usati più di frequente: possono essere registrati dinamicamente (il che significa che potete averne quanti ne volete), e garantiscono che un qualsiasi tasklet verrà eseguito solo su un processore alla volta, sebbene diversi tasklet possono essere eseguiti simultaneamente.

**Warning:** Il nome 'tasklet' è ingannevole: non hanno niente a che fare con i 'processi' ('tasks'), e probabilmente hanno più a che vedere con qualche pessima vodka che Alexey Kuznetsov si fece a quel tempo.

Potete determinate se siete in un softirq (o tasklet) utilizzando la macro `in_softirq()` (`include/linux/preempt.h`).

**Warning:** State attenti che questa macro ritornerà un falso positivo se `bottom half lock` è bloccato.

## Alcune regole basilari

**Nessuna protezione della memoria** Se corrompete la memoria, che sia in contesto utente o d'interruzione, la macchina si pianterà. Siete sicuri che quello che volete fare non possa essere fatto nello spazio utente?

**Nessun numero in virgola mobile o MMX** Il contesto della FPU non è salvato; anche se si ète in contesto utente lo stato dell'FPU probabilmente non corrisponde a quello del processo corrente: vi incasinerete con lo stato di qualche altro processo. Se volete davvero usare la virgola mobile, allora dovete salvare e recuperare lo stato dell'FPU (ed evitare cambi di contesto). Generalmente è una cattiva idea; usate l'aritmetica a virgola fissa.

**Un limite rigido dello stack** A seconda della configurazione del kernel lo stack è fra 3K e 6K per la maggior parte delle architetture a 32-bit; è di 14K per la maggior parte di quelle a 64-bit; e spesso è condiviso con le interruzioni, per cui non si può usare. Evitare profonde ricorsioni ad enormi array locali nello stack (allocateli dinamicamente).

**Il kernel Linux è portabile** Quindi mantenetelo tale. Il vostro codice dovrebbe essere a 64-bit ed indipendente dall'ordine dei byte (endianness) di un processore. Inoltre, dovreste minimizzare il codice specifico per un processore; per esempio il codice assembly dovrebbe essere encapsulato in modo pulito e minimizzato per facilitarne la migrazione. Generalmente questo codice dovrebbe essere limitato alla parte di kernel specifica per un'architettura.

## ioctl: non scrivere nuove chiamate di sistema

Una chiamata di sistema, generalmente, è scritta così:

```
asm linkage long sys_mycall(int arg)
{
    return 0;
}
```

Primo, nella maggior parte dei casi non volete creare nuove chiamate di sistema. Create un dispositivo a caratteri ed implementate l'appropriata chiamata ioctl. Questo meccanismo è molto più flessibile delle chiamate di sistema: esso non dev'essere dichiarato in tutte le architetture nei file `include/asm/unistd.h` e `arch/kernel/entry.S`; inoltre, è improbabile che questo venga accettato da Linus.

Se tutto quello che il vostro codice fa è leggere o scrivere alcuni parametri, considerate l'implementazione di un'interfaccia `sysfs()`.

All'interno di una ioctl vi trovate nel contesto utente di un processo. Quando avviene un errore dovete ritornare un valore negativo di `errno` (consultate `include/uapi/asm-generic/errno-base.h`, `include/uapi/asm-generic/errno.h` e `include/linux/errno.h`), altrimenti ritornate 0.

Dopo aver dormito dovreste verificare se ci sono stati dei segnali: il modo Unix/Linux di gestire un segnale è di uscire temporaneamente dalla chiamata di sistema con l'errore `-ERESTARTSYS`. La chiamata di sistema ritornerà al contesto utente, eseguirà il gestore del segnale e poi la vostra chiamata di sistema riprenderà (a meno che l'utente non l'abbia disabilitata). Quindi, dovreste essere pronti per continuare l'esecuzione, per esempio nel mezzo della manipolazione di una struttura dati.

```
if (signal_pending(current))
    return -ERESTARTSYS;
```

Se dovete eseguire dei calcoli molto lunghi: pensate allo spazio utente. Se **davvero** volete farlo nel kernel ricordatevi di verificare periodicamente se dovete *lasciare* il processore (ricordatevi che, per ogni processore, c'è un sistema multi-processo senza diritto di prelazione). Esempio:

```
cond_resched(); /* Will sleep */
```

Una breve nota sulla progettazione delle interfacce: il motto dei sistemi UNIX è “fornite meccanismi e non politiche”

### La ricetta per uno stallo

Non è permesso invocare una procedura che potrebbe dormire, fanno eccezione i seguenti casi:

- Siete in un contesto utente.
- Non trattenete alcun spinlock.
- Avete abilitato le interruzioni (in realtà, Andy Kleen dice che lo scheduler le abiliterà per voi, ma probabilmente questo non è quello che volete).

Da tener presente che alcune funzioni potrebbero dormire implicitamente: le più comuni sono quelle per l'accesso allo spazio utente (\*\_user) e quelle per l'allocazione della memoria senza l'opzione GFP\_ATOMIC

Dovreste sempre compilare il kernel con l'opzione CONFIG\_DEBUG\_ATOMIC\_SLEEP attiva, questa vi avviserà se infrangrete una di queste regole. Se **infrangete** le regole, allora potreste bloccare il vostro scatolotto.

Veramente.

### Alcune delle procedure più comuni

#### printk()

Definita in `include/linux/printk.h`

`printk()` fornisce messaggi alla console, dmesg, e al demone syslog. Essa è utile per il debugging o per la notifica di errori; può essere utilizzata anche all'interno del contesto d'interruzione, ma usatela con cautela: una macchina che ha la propria console inondata da messaggi diventa inutilizzabile. La funzione utilizza un formato stringa quasi compatibile con la `printf` ANSI C, e la concatenazione di una stringa C come primo argomento per indicare la “priorità”:

```
printk(KERN_INFO "i = %u\n", i);
```

Consultate `include/linux/kern_levels.h` per gli altri valori KERN\_ ; questi sono interpretati da syslog come livelli. Un caso speciale: per stampare un indirizzo IP usate:

```
_be32 ipaddress;
printk(KERN_INFO "my ip: %pI4\n", &ipaddress);
```

`printk()` utilizza un buffer interno di 1K e non s'accorge di eventuali sforamenti. Accertatevi che vi basti.

**Note:** Saprete di essere un vero hacker del kernel quando inizierete a digitare nei vostri programmi utenti le `printf` come se fossero `printk` :)

**Note:** Un'altra nota a parte: la versione originale di Unix 6 aveva un commento sopra alla funzione `printf`: "Printf non dovrebbe essere usata per il chiacchiericcio". Dovreste seguire questo consiglio.

### `copy_to_user()` / `copy_from_user()` / `get_user()` / `put_user()`

Definite in `include/linux/uaccess.h` / `asm/uaccess.h`

#### [DORMONO]

`put_user()` e `get_user()` sono usate per ricevere ed impostare singoli valori (come `int`, `char`, o `long`) da e verso lo spazio utente. Un puntatore nello spazio utente non dovrebbe mai essere dereferenziato: i dati dovrebbero essere copiati usando suddette procedure. Entrambe ritornano `-EFAULT` oppure `0`.

`copy_to_user()` e `copy_from_user()` sono più generiche: esse copiano una quantità arbitraria di dati da e verso lo spazio utente.

**Warning:** Al contrario di `c:func:put_user()` e `get_user()`, queste funzioni ritornano la quantità di dati copiati (0 è comunque un successo).

[Sì, questa stupida interfaccia mi imbarazza. La battaglia torna in auge anno dopo anno. -RR]

Le funzioni potrebbero dormire implicitamente. Queste non dovrebbero mai essere invocate fuori dal contesto utente (non ha senso), con le interruzioni disabilitate, o con uno spinlock trattenuto.

### `kmalloc()`/`kfree()`

Definite in `include/linux/slab.h`

#### [POTREBBERO DORMIRE: LEGGI SOTTO]

Queste procedure sono utilizzate per la richiesta dinamica di un puntatore ad un pezzo di memoria allineato, esattamente come `malloc` e `free` nello spazio utente, ma `kmalloc()` ha un argomento aggiuntivo per indicare alcune opzioni. Le opzioni più importanti sono:

**GFP\_KERNEL** Potrebbe dormire per liberare della memoria. L'opzione fornisce il modo più affidabile per allocare memoria, ma il suo uso è strettamente limitato allo spazio utente.

**GFP\_ATOMIC** Non dorme. Meno affidabile di **GFP\_KERNEL**, ma può essere usata in un contesto d'interruzione. Dovreste avere **davvero** una buona strategia per la gestione degli errori in caso di mancanza di memoria.

**GFP\_DMA** Alloca memoria per il DMA sul bus ISA nello spazio d'indirizzamento inferiore ai 16MB.  
Se non sapete cos'è allora non vi serve. Molto inaffidabile.

Se vedete un messaggio d'avviso per una funzione dormiente che viene chiamata da un contesto errato, allora probabilmente avete usato una funzione d'allocazione dormiente da un contesto d'interruzione senza **GFP\_ATOMIC**. Dovreste correggerlo. Sbrigatevi, non cincischiate.

Se allocate almeno `PAGE_SIZE``(``asm/page.h o asm/page_types.h)` byte, considerate l'uso di `__get_free_pages()` (`include/linux/gfp.h`). Accetta un argomento che definisce l'ordine (0 per la dimensione di una pagina, 1 per una doppia pagina, 2 per quattro pagine, eccetera) e le stesse opzioni d'allocazione viste precedentemente.

Se state allocando un numero di byte notevolemente superiore ad una pagina potete usare `vmalloc()`. Essa allocherà memoria virtuale all'interno dello spazio kernel. Questo è un blocco di memoria fisica non contiguo, ma la MMU vi darà l'impressione che lo sia (quindi, sarà contiguo solo dal punto di vista dei processori, non dal punto di vista dei driver dei dispositivi esterni). Se per qualche strana ragione avete davvero bisogno di una grossa quantità di memoria fisica contigua, avete un problema: Linux non ha un buon supporto per questo caso d'uso perché, dopo un po' di tempo, la frammentazione della memoria rende l'operazione difficile. Il modo migliore per allocare un simile blocco all'inizio dell'avvio del sistema è attraverso la procedura `alloc_bootmem()`.

Prima di inventare la vostra cache per gli oggetti più usati, considerate l'uso di una cache slab disponibile in `include/linux/slab.h`.

### current

Definita in `include/asm/current.h`

Questa variabile globale (in realtà una macro) contiene un puntatore alla struttura del processo corrente, quindi è valido solo dal contesto utente. Per esempio, quando un processo esegue una chiamata di sistema, questo punterà alla struttura dati del processo chiamante. Nel contesto d'interruzione in suo valore **non è NULL**.

### mdelay() / udelay()

Definite in `include/asm/delay.h` / `include/linux/delay.h`

Le funzioni `udelay()` e `ndelay()` possono essere utilizzate per brevi pause. Non usate grandi valori perché rischiate d'avere un overflow - in questo contesto la funzione `mdelay()` è utile, oppure considerate `msleep()`.

## `cpu_to_be32() / be32_to_cpu() / cpu_to_le32() / le32_to_cpu()`

Definite in `include/asm/byteorder.h`

La famiglia di funzioni `cpu_to_be32()` (dove “32” può essere sostituito da 64 o 16, e “be” con “le”) forniscono un modo generico per fare conversioni sull’ordine dei byte (endianness): esse ritornano il valore convertito. Tutte le varianti supportano anche il processo inverso: `be32_to_cpu()`, eccetera.

Queste funzioni hanno principalmente due varianti: la variante per puntatori, come `cpu_to_be32p()`, che prende un puntatore ad un tipo, e ritorna il valore convertito. L’altra variante per la famiglia di conversioni “in-situ”, come `cpu_to_be32s()`, che convertono il valore puntato da un puntatore, e ritornano void.

## `local_irq_save() / local_irq_restore()`

Definite in `include/linux/irqflags.h`

Queste funzioni abilitano e disabilitano le interruzioni hardware sul processore locale. Entrambe sono rientranti; esse salvano lo stato precedente nel proprio argomento `unsigned long flags`. Se sapete che le interruzioni sono abilitate, potete semplicemente utilizzare `local_irq_disable()` e `local_irq_enable()`.

## `local_bh_disable() / local_bh_enable()`

Definite in `include/linux/bottom_half.h`

Queste funzioni abilitano e disabilitano le interruzioni software sul processore locale. Entrambe sono rientranti; se le interruzioni software erano già state disabilitate in precedenza, rimarranno disabilitate anche dopo aver invocato questa coppia di funzioni. Lo scopo è di prevenire l’esecuzione di softirq e tasklet sul processore attuale.

## `smp_processor_id()`

Definita in `include/linux/smp.h`

`get_cpu()` nega il diritto di prelazione (quindi non potete essere spostati su un altro processore all’improvviso) e ritorna il numero del processore attuale, fra 0 e `NR_CPUS`. Da notare che non è detto che la numerazione dei processori sia continua. Quando avete terminato, ritornate allo stato precedente con `put_cpu()`.

Se sapete che non dovete essere interrotti da altri processi (per esempio, se siete in un contesto d’interruzione, o il diritto di prelazione è disabilitato) potete utilizzare `smp_processor_id()`.

### `__init/ __exit/ __initdata`

Definite in `include/linux/init.h`

Dopo l'avvio, il kernel libera una sezione speciale; le funzioni marcate con `__init` e le strutture dati marcate con `__initdata` vengono eliminate dopo il completamento dell'avvio: in modo simile i moduli eliminano questa memoria dopo l'inizializzazione. `__exit` viene utilizzato per dichiarare che una funzione verrà utilizzata solo in fase di rimozione: la detta funzione verrà eliminata quando il file che la contiene non è compilato come modulo. Guardate l'header file per informazioni. Da notare che non ha senso avere una funzione marcata come `__init` e al tempo stesso esportata ai moduli utilizzando `EXPORT_SYMBOL()` o `EXPORT_SYMBOL_GPL()` - non funzionerà.

### `__initcall()/module_init()`

Definite in `include/linux/init.h / include/linux/module.h`

Molte parti del kernel funzionano bene come moduli (componenti del kernel caricabili dinamicamente). L'utilizzo delle macro `module_init()` e `module_exit()` semplifica la scrittura di codice che può funzionare sia come modulo, sia come parte del kernel, senza l'ausilio di `#ifdef`.

La macro `module_init()` definisce quale funzione dev'essere chiamata quando il modulo viene inserito (se il file è stato compilato come tale), o in fase di avvio : se il file non è stato compilato come modulo la macro `module_init()` diventa equivalente a `__initcall()`, la quale, tramite qualche magia del linker, s'assicura che la funzione venga chiamata durante l'avvio.

La funzione può ritornare un numero d'errore negativo per scatenare un fallimento del caricamento (sfortunatamente, questo non ha effetto se il modulo è compilato come parte integrante del kernel). Questa funzione è chiamata in contesto utente con le interruzioni abilitate, quindi potrebbe dormire.

### `module_exit()`

Definita in `include/linux/module.h`

Questa macro definisce la funzione che dev'essere chiamata al momento della rimozione (o mai, nel caso in cui il file sia parte integrante del kernel). Essa verrà chiamata solo quando il contatore d'uso del modulo raggiunge lo zero. Questa funzione può anche dormire, ma non può fallire: tutto dev'essere ripulito prima che la funzione ritorni.

Da notare che questa macro è opzionale: se non presente, il modulo non sarà removibile (a meno che non usiate '`rmmmod -f`' ).

## try\_module\_get()/module\_put()

Definite in `include/linux/module.h`

Queste funzioni maneggiano il contatore d'uso del modulo per proteggerlo dalla rimozione (in aggiunta, un modulo non può essere rimosso se un altro modulo utilizza uno dei suoi simboli esportati: vedere di seguito). Prima di eseguire codice del modulo, dovreste chiamare `try_module_get()` su quel modulo: se fallisce significa che il modulo è stato rimosso e dovete agire come se non fosse presente. Altrimenti, potete accedere al modulo in sicurezza, e chiamare `module_put()` quando avete finito.

La maggior parte delle strutture registrabili hanno un campo `owner` (proprietario), come nella struttura `struct file_operations`. Impostate questo campo al valore della macro `THIS_MODULE`.

## Code d'attesa `include/linux/wait.h`

### [DORMONO]

Una coda d'attesa è usata per aspettare che qualcuno vi attivi quando una certa condizione s'avvera. Per evitare corse critiche, devono essere usate con cautela. Dichiaretene una `wait_queue_head_t`, e poi i processi che vogliono attendere il verificarsi di quella condizione dichiareranno una `wait_queue_entry_t` facendo riferimento a loro stessi, poi metteranno questa in coda.

### Dichiarazione

Potere dichiarare una `wait_queue_head_t` utilizzando la macro `DECLARE_WAIT_QUEUE_HEAD()` oppure utilizzando la procedura `init_waitqueue_head()` nel vostro codice d'inizializzazione.

### Accodamento

Mettersi in una coda d'attesa è piuttosto complesso, perché dovete mettervi in coda prima di verificare la condizione. Esiste una macro a questo scopo: `wait_event_interruptible()` (`include/linux/wait.h`). Il primo argomento è la testa della coda d'attesa, e il secondo è un'espressione che dev'essere valutata; la macro ritorna 0 quando questa espressione è vera, altrimenti -ERESTARTSYS se è stato ricevuto un segnale. La versione `wait_event()` ignora i segnali.

### Svegliare una procedura in coda

Chiamate `wake_up()` (`include/linux/wait.h`); questa attiverà tutti i processi in coda. Ad eccezione se uno di questi è impostato come `TASK_EXCLUSIVE`, in questo caso i rimanenti non verranno svegliati. Nello stesso header file esistono altre varianti di questa funzione.

### Operazioni atomiche

Certe operazioni sono garantite come atomiche su tutte le piattaforme. Il primo gruppo di operazioni utilizza `atomic_t` (`include/asm/atomic.h`); questo contiene un intero con segno (minimo 32bit), e dovete utilizzare queste funzione per modificare o leggere variabili di tipo `atomic_t`. `atomic_read()` e `atomic_set()` leggono ed impostano il contatore, `atomic_add()`, `atomic_sub()`, `atomic_inc()`, `atomic_dec()`, e `atomic_dec_and_test()` (ritorna vero se raggiunge zero dopo essere stata decrementata).

Sì. Ritorna vero (ovvero `!= 0`) se la variabile atomica è zero.

Da notare che queste funzioni sono più lente rispetto alla normale aritmetica, e quindi non dovrebbero essere usate a sproposito.

Il secondo gruppo di operazioni atomiche sono definite in `include/linux/bitops.h` ed agiscono sui bit d'una variabile di tipo `unsigned long`. Queste operazioni prendono come argomento un puntatore alla variabile, e un numero di bit dove 0 è quello meno significativo. `set_bit()`, `clear_bit()` e `change_bit()` impostano, cancellano, ed invertono il bit indicato. `test_and_set_bit()`, `test_and_clear_bit()` e `test_and_change_bit()` fanno la stessa cosa, ad eccezione che ritornano vero se il bit era impostato; queste sono particolarmente utili quando si vuole impostare atomicamente dei flag.

Con queste operazioni è possibile utilizzare indici di bit che eccedono il valore `BITS_PER_LONG`. Il comportamento è strano sulle piattaforme big-endian quindi è meglio evitarlo.

### Simboli

All'interno del kernel, si seguono le normali regole del linker (ovvero, a meno che un simbolo non venga dichiarato con visibilità limitata ad un file con la parola chiave `static`, esso può essere utilizzato in qualsiasi parte del kernel). Nonostante ciò, per i moduli, esiste una tabella dei simboli esportati che limita i punti di accesso al kernel. Anche i moduli possono esportare simboli.

#### `EXPORT_SYMBOL()`

Definita in `include/linux/export.h`

Questo è il classico metodo per esportare un simbolo: i moduli caricati dinamicamente potranno utilizzare normalmente il simbolo.

#### `EXPORT_SYMBOL_GPL()`

Definita in `include/linux/export.h`

Essa è simile a `EXPORT_SYMBOL()` ad eccezione del fatto che i simboli esportati con `EXPORT_SYMBOL_GPL()` possono essere utilizzati solo dai moduli che hanno dichiarato una licenza compatibile con la GPL attraverso `MODULE_LICENSE()`. Questo implica che la funzione esportata è considerata interna, e non una vera e propria interfaccia. Alcuni manutentori e sviluppatori potrebbero comunque richiedere `EXPORT_SYMBOL_GPL()` quando si aggiungono nuove funzionalità o interfacce.

## **EXPORT\_SYMBOL\_NS()**

Definita in `include/linux/export.h`

Questa è una variante di `EXPORT_SYMBOL()` che permette di specificare uno spazio dei nomi. Lo spazio dei nomi è documentato in [Spazio dei nomi dei simboli](#).

## **EXPORT\_SYMBOL\_NS\_GPL()**

Definita in `include/linux/export.h`

Questa è una variante di `EXPORT_SYMBOL_GPL()` che permette di specificare uno spazio dei nomi. Lo spazio dei nomi è documentato in [Spazio dei nomi dei simboli](#).

## **Procedure e convenzioni**

### **Liste doppiamente concatenate `include/linux/list.h`**

Un tempo negli header del kernel c'erano tre gruppi di funzioni per le liste concatenate, ma questa è stata la vincente. Se non avete particolari necessità per una semplice lista concatenata, allora questa è una buona scelta.

In particolare, `list_for_each_entry()` è utile.

### **Convenzione dei valori di ritorno**

Per codice chiamato in contesto utente, è molto comune sfidare le convenzioni C e ritornare 0 in caso di successo, ed un codice di errore negativo (eg. `-EFAULT`) nei casi fallimentari. Questo potrebbe essere controiduitivo a prima vista, ma è abbastanza diffuso nel kernel.

Utilizzate `ERR_PTR()` (`include/linux/err.h`) per codificare un numero d'errore negativo in un puntatore, e `IS_ERR()` e `PTR_ERR()` per recuperarlo di nuovo: così si evita d'avere un puntatore dedicato per il numero d'errore. Da brividi, ma in senso positivo.

### **Rompere la compilazione**

Linus e gli altri sviluppatori a volte cambiano i nomi delle funzioni e delle strutture nei kernel in sviluppo; questo non è solo per tenere tutti sulle spine: questo riflette cambiamenti fondamentali (eg. la funzione non può più essere chiamata con le funzioni attive, o fa controlli aggiuntivi, o non fa più controlli che venivano fatti in precedenza). Solitamente a questo s'accompagna un'adeguata e completa nota sulla lista di discussione linux-kernel; cercate negli archivi. Solitamente eseguire una semplice sostituzione su tutto un file rendere le cose **peggiori**.

### Inizializzazione dei campi d'una struttura

Il metodo preferito per l'inizializzazione delle strutture è quello di utilizzare gli inizializzatori designati, come definiti nello standard ISO C99, eg:

```
static struct block_device_operations opt_fops = {
    .open          = opt_open,
    .release       = opt_release,
    .ioctl         = opt_ioctl,
    .check_media_change = opt_media_change,
};
```

Questo rende più facile la ricerca con grep, e rende più chiaro quale campo viene impostato. Dovreste fare così perché si mostra meglio.

### Estensioni GNU

Le estensioni GNU sono esplicitamente permesse nel kernel Linux. Da notare che alcune delle più complesse non sono ben supportate, per via dello scarso sviluppo, ma le seguenti sono da considerarsi la norma (per maggiori dettagli, leggete la sezione "C Extensions" nella pagina info di GCC - Sì, davvero la pagina info, la pagina man è solo un breve riassunto delle cose nella pagina info).

- Funzioni inline
- Istruzioni in espressioni (ie. il costrutto ({ and }) ).
- Dichiarate attributi di una funzione / variabile / tipo (`_attribute_`)
- `typeof`
- Array con lunghezza zero
- Macro varargs
- Aritmetica sui puntatori void
- Inizializzatori non costanti
- Istruzioni assembler (non al di fuori di 'arch/' e 'include/asm/')
- Nomi delle funzioni come stringhe (`_func_`).
- `_builtin_constant_p()`

Siate sospettosi quando utilizzate long long nel kernel, il codice generato da gcc è orribile ed anche peggio: le divisioni e le moltiplicazioni non funzionano sulle piattaforme i386 perché le rispettive funzioni di runtime di GCC non sono incluse nell'ambiente del kernel.

## C++

Soltamente utilizzare il C++ nel kernel è una cattiva idea perché il kernel non fornisce il necessario ambiente di runtime e gli header file non sono stati verificati. Rimane comunque possibile, ma non consigliato. Se davvero volete usarlo, almeno evitate le eccezioni.

## NUMif

Viene generalmente considerato più pulito l'uso delle macro negli header file (o all'inizio dei file .c) per astrarre funzioni piuttosto che utilizzare l'istruzione di pre-processore `#if` all'interno del codice sorgente.

## Mettere le vostre cose nel kernel

Al fine d'avere le vostre cose in ordine per l'inclusione ufficiale, o anche per avere patch pulite, c'è del lavoro amministrativo da fare:

- Trovare di chi è lo stagno in cui state pisciando. Guardare in cima ai file sorgenti, all'interno del file MAINTAINERS, ed alla fine di tutti nel file CREDITS. Dovreste coordinarvi con queste persone per evitare di duplicare gli sforzi, o provare qualcosa che è già stato rigettato.

Assicuratevi di mettere il vostro nome ed indirizzo email in cima a tutti i file che create o che mangeggiate significativamente. Questo è il primo posto dove le persone guarderanno quando troveranno un baco, o quando **loro** vorranno fare una modifica.

- Solitamente vorrete un'opzione di configurazione per la vostra modifica al kernel. Modificate Kconfig nella cartella giusta. Il linguaggio Config è facile con copia ed incolla, e c'è una completa documentazione nel file Documentation/kbuild/kconfig-language.rst.

Nella descrizione della vostra opzione, assicuratevi di parlare sia agli utenti esperti sia agli utenti che non sanno nulla del vostro lavoro. Menzionate qui le incompatibilità ed i problemi. Chiaramente la descrizione deve terminare con "if in doubt, say N" (se siete in dubbio, dite N) (oppure, occasionalmente, 'Y'); questo è per le persone che non hanno idea di che cosa voi stiate parlando.

- Modificate il file Makefile: le variabili CONFIG sono esportate qui, quindi potete solitamente aggiungere una riga come la seguente "obj-\$(CONFIG\_xxx) += xxx.o". La sintassi è documentata nel file Documentation/kbuild/makefiles.rst.
- Aggiungete voi stessi in CREDITS se avete fatto qualcosa di notevole, solitamente qualcosa che supera il singolo file (comunque il vostro nome dovrebbe essere all'inizio dei file sorgenti). MAINTAINERS significa che volete essere consultati quando vengono fatte delle modifiche ad un sottosistema, e quando ci sono dei bachi; questo implica molto di più di un semplice impegno su una parte del codice.
- Infine, non dimenticatevi di leggere Documentation/process/submitting-patches.rst e possibilmente anche Documentation/process/submitting-drivers.rst.

### Trucchetti del kernel

Dopo una rapida occhiata al codice, questi sono i preferiti. Sentitevi liberi di aggiungerne altri.

arch/x86/include/asm/delay.h:

```
#define ndelay(n)  (_builtin_constant_p(n) ? \  
                  ((n) > 20000 ? __bad_ndelay() : __const_udelay((n) * 5ul)) : \  
                  __ndelay(n))
```

include/linux/fs.h:

```
/*  
 * Kernel pointers have redundant information, so we can use a  
 * scheme where we can return either an error code or a dentry  
 * pointer with the same return value.  
 *  
 * This should be a per-architecture thing, to allow different  
 * error and pointer decisions.  
 */  
#define ERR_PTR(err)      ((void *)((long)(err)))  
#define PTR_ERR(ptr)     ((long)(ptr))  
#define IS_ERR(ptr)       ((unsigned long)(ptr) > (unsigned long)(-1000))
```

arch/x86/include/asm/uaccess\_32.h::

```
#define copy_to_user(to,from,n)           \  
    (_builtin_constant_p(n) ?             \  
    __constant_copy_to_user((to),(from),(n)) :   \  
    __generic_copy_to_user((to),(from),(n)))
```

arch/sparc/kernel/head.S::

```
/*  
 * Sun people can't spell worth damn. "compatability" indeed.  
 * At least we *know* we can't spell, and use a spell-checker.  
 */  
  
/* Uh, actually Linus it is I who cannot spell. Too much murky  
 * Sparc assembly will do this to ya.  
 */  
C_LABEL(cputypvar):  
    .asciz "compatibility"  
  
/* Tested on SS-5, SS-10. Probably someone at Sun applied a spell-checker. */  
    .align 4  
C_LABEL(cputypvar_sun4m):  
    .asciz "compatible"
```

arch/sparc/lib/checksum.S::

```
/* Sun, you just can't beat me, you just can't. Stop trying,
 * give up. I'm serious, I am going to kick the living shit
 * out of you, game over, lights out.
 */
```

## Ringraziamenti

Ringrazio Andi Kleen per le sue idee, le risposte alle mie domande, le correzioni dei miei errori, l'aggiunta di contenuti, eccetera. Philipp Rumpf per l'ortografia e per aver reso più chiaro il testo, e per alcuni eccellenti punti tutt'altro che ovvi. Werner Almesberger per avermi fornito un ottimo riassunto di `disable_irq()`, e Jes Sorensen e Andrea Arcangeli per le precisazioni. Michael Elizabeth Chastain per aver verificato ed aggiunto la sezione configurazione. Telsa Gwynne per avermi insegnato DocBook.

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** Documentation/kernel-hacking/locking.rst

**Translator** Federico Vaga <[federico.vaga@vaga.pv.it](mailto:federico.vaga@vaga.pv.it)>

## L'inaffidabile guida alla sincronizzazione

**Author** Rusty Russell

### Introduzione

Benvenuto, alla notevole ed inaffidabile guida ai problemi di sincronizzazione (locking) nel kernel. Questo documento descrive il sistema di sincronizzazione nel kernel Linux 2.6.

Dato il largo utilizzo del multi-threading e della prelazione nel kernel Linux, chiunque voglia dilettarsi col kernel deve conoscere i concetti fondamentali della concorrenza e della sincronizzazione nei sistemi multi-processore.

### Il problema con la concorrenza

(Saltatelo se sapete già cos'è una corsa critica).

In un normale programma, potete incrementare un contatore nel seguente modo:

```
contatore++;
```

Questo è quello che vi aspettereste che accada sempre:

Table 3: Risultati attesi

Istanza 1	Istanza 2
leggi contatore (5)	
aggiungi 1 (6)	
scrivi contatore (6)	
	leggi contatore (6)
	aggiungi 1 (7)
	scrivi contatore (7)

Questo è quello che potrebbe succedere in realtà:

Table 4: Possibile risultato

Istanza 1	Istanza 2
leggi contatore (5)	
	leggi contatore (5)
aggiungi 1 (6)	
	aggiungi 1 (6)
scrivi contatore (6)	
	scrivi contatore (6)

## Corse critiche e sezioni critiche

Questa sovrapposizione, ovvero quando un risultato dipende dal tempo che intercorre fra processi diversi, è chiamata corsa critica. La porzione di codice che contiene questo problema è chiamata sezione critica. In particolar modo da quando Linux ha incominciato a girare su macchine multi-processore, le sezioni critiche sono diventate uno dei maggiori problemi di progettazione ed implementazione del kernel.

La prelazione può sortire gli stessi effetti, anche se c'è una sola CPU: interrompendo un processo nella sua sezione critica otterremo comunque la stessa corsa critica. In questo caso, il thread che si avvicenda nell'esecuzione potrebbe eseguire anch'esso la sezione critica.

La soluzione è quella di riconoscere quando avvengono questi accessi simultanei, ed utilizzare i *lock* per accertarsi che solo un'istanza per volta possa entrare nella sezione critica. Il kernel offre delle buone funzioni a questo scopo. E poi ci sono quelle meno buone, ma farò finta che non esistano.

## Sincronizzazione nel kernel Linux

Se posso darvi un suggerimento: non dormite mai con qualcuno più pazzo di voi. Ma se dovessi darvi un suggerimento sulla sincronizzazione: **mantenetela semplice**.

Siate riluttanti nell'introduzione di nuovi *lock*.

Abbastanza strano, quest'ultimo è l'esatto opposto del mio suggerimento su quando **avete** dormito con qualcuno più pazzo di voi. E dovreste pensare a prendervi un cane bello grande.

## I due principali tipi di *lock* nel kernel: spinlock e mutex

Ci sono due tipi principali di *lock* nel kernel. Il tipo fondamentale è lo spinlock (`include/asm/spinlock.h`), un semplice *lock* che può essere trattenuto solo da un processo: se non si può trattenere lo spinlock, allora rimane in attesa attiva (in inglese *spinning*) finché non ci riesce. Gli spinlock sono molto piccoli e rapidi, possono essere utilizzati ovunque.

Il secondo tipo è il mutex (`include/linux/mutex.h`): è come uno spinlock, ma potrete bloccarvi trattenendolo. Se non potete trattenere un mutex il vostro processo si auto-sosponderà; verrà riattivato quando il mutex verrà rilasciato. Questo significa che il processore potrà occuparsi d'altro mentre il vostro processo è in attesa. Esistono molti casi in cui non potete permettervi di sospendere un processo (vedere [Quali funzioni possono essere chiamate in modo sicuro dalle interruzioni?](#)) e quindi dovete utilizzare gli spinlock.

Nessuno di questi *lock* è ricorsivo: vedere [Stallo: semplice ed avanzato](#)

## I *lock* e i kernel per sistemi monoprocesso

Per i kernel compilati senza `CONFIG_SMP` e senza `CONFIG_PREEMPT` gli spinlock non esistono. Questa è un'ottima scelta di progettazione: quando nessun altro processo può essere eseguito in simultanea, allora non c'è la necessità di avere un *lock*.

Se il kernel è compilato senza `CONFIG_SMP` ma con `CONFIG_PREEMPT`, allora gli spinlock disabilitano la prelazione; questo è sufficiente a prevenire le corse critiche. Nella maggior parte dei casi, possiamo considerare la prelazione equivalente ad un sistema multi-processore senza preoccuparci di trattarla indipendentemente.

Dovreste verificare sempre la sincronizzazione con le opzioni `CONFIG_SMP` e `CONFIG_PREEMPT` abilitate, anche quando non avete un sistema multi-processore, questo vi permetterà di identificare alcuni problemi di sincronizzazione.

Come vedremo di seguito, i mutex continuano ad esistere perché sono necessari per la sincronizzazione fra processi in contesto utente.

## Sincronizzazione in contesto utente

Se avete una struttura dati che verrà utilizzata solo dal contesto utente, allora, per proteggerla, potete utilizzare un semplice mutex (`include/linux/mutex.h`). Questo è il caso più semplice: inizializzate il mutex; invocate `mutex_lock_interruptible()` per trattenerlo e `mutex_unlock()` per rilasciarlo. C'è anche `mutex_lock()` ma questa dovrebbe essere evitata perché non ritorna in caso di segnali.

Per esempio: `net/netfilter/nf_sockopt.c` permette la registrazione di nuove chiamate per `setsockopt()` e `getsockopt()` usando la funzione `nf_register_sockopt()`. La registrazione e la rimozione vengono eseguite solamente quando il modulo viene caricato o scaricato (e durante l'avvio del sistema, qui non abbiamo concorrenza), e la lista delle funzioni registrate viene consultata solamente quando `setsockopt()` o `getsockopt()` sono sconosciute al sistema. In questo caso `nf_sockopt_mutex` è perfetto allo scopo, in particolar modo visto che `setsockopt` e `getsockopt` potrebbero dormire.

### Sincronizzazione fra il contesto utente e i softirq

Se un softirq condivide dati col contesto utente, avete due problemi. Primo, il contesto utente corrente potrebbe essere interrotto da un softirq, e secondo, la sezione critica potrebbe essere eseguita da un altro processore. Questo è quando `spin_lock_bh()` (`include/linux/spinlock.h`) viene utilizzato. Questo disabilita i softirq sul processore e trattiene il *lock*. Invece, `spin_unlock_bh()` fa l'opposto. (Il suffisso '`_bh`' è un residuo storico che fa riferimento al "Bottom Halves", il vecchio nome delle interruzioni software. In un mondo perfetto questa funzione si chiamerebbe '`spin_lock_softirq()`').

Da notare che in questo caso potete utilizzare anche `spin_lock_irq()` o `spin_lock_irqsave()`, queste fermano anche le interruzioni hardware: vedere [Contesto di interruzione hardware](#).

Questo funziona alla perfezione anche sui sistemi monoprocesso: gli spinlock svaniscono e questa macro diventa semplicemente `local_bh_disable()` (`include/linux/interrupt.h`), la quale impedisce ai softirq d'essere eseguiti.

### Sincronizzazione fra contesto utente e i tasklet

Questo caso è uguale al precedente, un tasklet viene eseguito da un softirq.

### Sincronizzazione fra contesto utente e i timer

Anche questo caso è uguale al precedente, un timer viene eseguito da un softirq. Dal punto di vista della sincronizzazione, tasklet e timer sono identici.

### Sincronizzazione fra tasklet e timer

Qualche volta un tasklet od un timer potrebbero condividere i dati con un altro tasklet o timer

#### Lo stesso tasklet/timer

Dato che un tasklet non viene mai eseguito contemporaneamente su due processori, non dovete preoccuparvi che sia rientrante (ovvero eseguito più volte in contemporanea), perfino su sistemi multi-processore.

#### Differenti tasklet/timer

Se un altro tasklet/timer vuole condividere dati col vostro tasklet o timer, allora avrete bisogno entrambe di `spin_lock()` e `spin_unlock()`. Qui `spin_lock_bh()` è inutile, siete già in un tasklet ed avete la garanzia che nessun altro verrà eseguito sullo stesso processore.

## Sincronizzazione fra softirq

Spesso un softirq potrebbe condividere dati con se stesso o un tasklet/timer.

### Lo stesso softirq

Lo stesso softirq può essere eseguito su un diverso processore: allo scopo di migliorare le prestazioni potete utilizzare dati riservati ad ogni processore (vedere [Dati per processore](#)). Se siete arrivati fino a questo punto nell'uso dei softirq, probabilmente tenete alla scalabilità delle prestazioni abbastanza da giustificare la complessità aggiuntiva.

Dovete utilizzare `spin_lock()` e `spin_unlock()` per proteggere i dati condivisi.

### Diversi Softirqs

Dovete utilizzare `spin_lock()` e `spin_unlock()` per proteggere i dati condivisi, che siano timer, tasklet, diversi softirq o lo stesso o altri softirq: uno qualsiasi di essi potrebbe essere in esecuzione su un diverso processore.

### Contesto di interruzione hardware

Solitamente le interruzioni hardware comunicano con un tasklet o un softirq. Spesso questo si traduce nel mettere in coda qualcosa da fare che verrà preso in carico da un softirq.

### Sincronizzazione fra interruzioni hardware e softirq/tasklet

Se un gestore di interruzioni hardware condivide dati con un softirq, allora avrete due preoccupazioni. Primo, il softirq può essere interrotto da un'interruzione hardware, e secondo, la sezione critica potrebbe essere eseguita da un'interruzione hardware su un processore diverso. Questo è il caso dove `spin_lock_irq()` viene utilizzato. Disabilita le interruzioni sul processore che l'esegue, poi trattiene il lock. `spin_unlock_irq()` fa l'opposto.

Il gestore d'interruzione hardware non ha bisogno di usare `spin_lock_irq()` perché i softirq non possono essere eseguiti quando il gestore d'interruzione hardware è in esecuzione: per questo si può usare `spin_lock()`, che è un po' più veloce. L'unica eccezione è quando un altro gestore d'interruzioni hardware utilizza lo stesso *lock*: `spin_lock_irq()` impedirà a questo secondo gestore di interrompere quello in esecuzione.

Questo funziona alla perfezione anche sui sistemi monoprocesso: gli spinlock svaniscono e questa macro diventa semplicemente `local_irq_disable()` (`include/asm/smp.h`), la quale impedisce a softirq/tasklet/BH d'essere eseguiti.

`spin_lock_irqsave()` (`include/linux/spinlock.h`) è una variante che salva lo stato delle interruzioni in una variabile, questa verrà poi passata a `spin_unlock_irqrestore()`. Questo significa che lo stesso codice potrà essere utilizzato in un'interruzione hardware (dove le interruzioni sono già disabilitate) e in un softirq (dove la disabilitazione delle interruzioni è richiesta).

Da notare che i softirq (e quindi tasklet e timer) sono eseguiti al ritorno da un'interruzione hardware, quindi `spin_lock_irq()` interrompe anche questi. Tenuto conto di questo si può dire che `spin_lock_irqsave()` è la funzione di sincronizzazione più generica e potente.

### Sincronizzazione fra due gestori d'interruzioni hardware

Condividere dati fra due gestori di interruzione hardware è molto raro, ma se succede, dovreste usare `spin_lock_irqsave()`: è una specificità dell'architettura il fatto che tutte le interruzioni vengano interrotte quando si eseguono di gestori di interruzioni.

### Bigino della sincronizzazione

Pete Zaitcev ci offre il seguente riassunto:

- Se siete in un contesto utente (una qualsiasi chiamata di sistema) e volete sincronizzarvi con altri processi, usate i mutex. Potete trattenere il mutex e dormire (`copy_from_user*`( o `kmalloc(x, GFP_KERNEL)`)).
- Altrimenti (== i dati possono essere manipolati da un'interruzione) usate `spin_lock_irqsave()` e `spin_unlock_irqrestore()`.
- Evitate di trattenere uno spinlock per più di 5 righe di codice incluse le chiamate a funzione (ad eccezione di quell per l'accesso come `readb()`).

### Tabella dei requisiti minimi

La tabella seguente illustra i requisiti **minimi** per la sincronizzazione fra diversi contesti. In alcuni casi, lo stesso contesto può essere eseguito solo da un processore per volta, quindi non ci sono requisiti per la sincronizzazione (per esempio, un thread può essere eseguito solo su un processore alla volta, ma se deve condividere dati con un altro thread, allora la sincronizzazione è necessaria).

Ricordatevi il suggerimento qui sopra: potete sempre usare `spin_lock_irqsave()`, che è un sovrainsieme di tutte le altre funzioni per spinlock.

.	IRQ Handler A	IRQ Handler B	Softirq A	Softirq B	Tasklet A	Tasklet B	Timer A	Timer B	User Context A	User Context B
IRQ Handler A	None									
IRQ Handler B	SLIS	None								
Softirq A	SLI	SLI	SL							
Softirq B	SLI	SLI	SL	SL						
Tasklet A	SLI	SLI	SL	SL	None					
Tasklet B	SLI	SLI	SL	SL	SL	None				
Timer A	SLI	SLI	SL	SL	SL	SL	None			
Timer B	SLI	SLI	SL	SL	SL	SL	SL	None		
User Context A	SLI	SLI	SLBH	SLBH	SLBH	SLBH	SLBH	SLBH	None	
User Context B	SLI	SLI	SLBH	SLBH	SLBH	SLBH	SLBH	SLBH	MLI	None

Table: Tabella dei requisiti per la sincronizzazione

SLIS	spin_lock_irqsave
SLI	spin_lock_irq
SL	spin_lock
SLBH	spin_lock_bh
MLI	mutex_lock_interruptible

Table: Legenda per la tabella dei requisiti per la sincronizzazione

## Le funzioni *trylock*

Ci sono funzioni che provano a trattenere un *lock* solo una volta e ritornano immediatamente comunicato il successo od il fallimento dell'operazione. Posso essere usate quando non serve accedere ai dati protetti dal *lock* quando qualche altro thread lo sta già facendo trattenendo il *lock*. Potrete acquisire il *lock* più tardi se vi serve accedere ai dati protetti da questo *lock*.

La funzione `spin_trylock()` non ritenta di acquisire il *lock*, se ci riesce al primo colpo ritorna un valore diverso da zero, altrimenti se fallisce ritorna 0. Questa funzione può essere utilizzata in un qualunque contesto, ma come `spin_lock()`: dovete disabilitare i contesti che potrebbero interrompervi e quindi trattenere lo spinlock.

La funzione `mutex_trylock()` invece di sospendere il vostro processo ritorna un valore diverso da zero se è possibile trattenere il lock al primo colpo, altrimenti se fallisce ritorna 0. Nonostante non dorma, questa funzione non può essere usata in modo sicuro in contesti di interruzione hardware o software.

### Esempi più comuni

Guardiamo un semplice esempio: una memoria che associa nomi a numeri. La memoria tiene traccia di quanto spesso viene utilizzato ogni oggetto; quando è piena, l'oggetto meno usato viene eliminato.

### Tutto in contesto utente

Nel primo esempio, supponiamo che tutte le operazioni avvengano in contesto utente (in soldoni, da una chiamata di sistema), quindi possiamo dormire. Questo significa che possiamo usare i mutex per proteggere la nostra memoria e tutti gli oggetti che contiene. Ecco il codice:

```
#include <linux/list.h>
#include <linux/slab.h>
#include <linux/string.h>
#include <linux/mutex.h>
#include <asm/errno.h>

struct object
{
    struct list_head list;
    int id;
    char name[32];
    int popularity;
};

/* Protects the cache, cache_num, and the objects within it */
static DEFINE_MUTEX(cache_lock);
static LIST_HEAD(cache);
static unsigned int cache_num = 0;
#define MAX_CACHE_SIZE 10

/* Must be holding cache_lock */
static struct object *_cache_find(int id)
{
    struct object *i;

    list_for_each_entry(i, &cache, list)
        if (i->id == id) {
            i->popularity++;
            return i;
        }
    return NULL;
}
```

```

/* Must be holding cache_lock */
static void __cache_delete(struct object *obj)
{
    BUG_ON(!obj);
    list_del(&obj->list);
    kfree(obj);
    cache_num--;
}

/* Must be holding cache_lock */
static void __cache_add(struct object *obj)
{
    list_add(&obj->list, &cache);
    if (++cache_num > MAX_CACHE_SIZE) {
        struct object *i, *outcast = NULL;
        list_for_each_entry(i, &cache, list) {
            if (!outcast || i->popularity < outcast->popularity)
                outcast = i;
        }
        __cache_delete(outcast);
    }
}

int cache_add(int id, const char *name)
{
    struct object *obj;

    if ((obj = kmalloc(sizeof(*obj), GFP_KERNEL)) == NULL)
        return -ENOMEM;

    strscpy(obj->name, name, sizeof(obj->name));
    obj->id = id;
    obj->popularity = 0;

    mutex_lock(&cache_lock);
    __cache_add(obj);
    mutex_unlock(&cache_lock);
    return 0;
}

void cache_delete(int id)
{
    mutex_lock(&cache_lock);
    __cache_delete(__cache_find(id));
    mutex_unlock(&cache_lock);
}

int cache_find(int id, char *name)
{

```

```

    struct object *obj;
    int ret = -ENOENT;

    mutex_lock(&cache_lock);
    obj = __cache_find(id);
    if (obj) {
        ret = 0;
        strcpy(name, obj->name);
    }
    mutex_unlock(&cache_lock);
    return ret;
}

```

Da notare che ci assicuriamo sempre di trattenere `cache_lock` quando aggiungiamo, rimuoviamo od ispezioniamo la memoria: sia la struttura della memoria che il suo contenuto sono protetti dal *lock*. Questo caso è semplice dato che copiamo i dati dall'utente e non permettiamo mai loro di accedere direttamente agli oggetti.

C'è una piccola ottimizzazione qui: nella funzione `cache_add()` impostiamo i campi dell'oggetto prima di acquisire il *lock*. Questo è sicuro perché nessun altro potrà accedervi finché non lo inseriremo nella memoria.

## Accesso dal contesto utente

Ora consideriamo il caso in cui `cache_find()` può essere invocata dal contesto d'interruzione: sia hardware che software. Un esempio potrebbe essere un timer che elimina oggetti dalla memoria.

Qui di seguito troverete la modifica nel formato *patch*: le righe - sono quelle rimosse, mentre quelle + sono quelle aggiunte.

```

--- cache.c.usercontext 2003-12-09 13:58:54.000000000 +1100
+++ cache.c.interrupt 2003-12-09 14:07:49.000000000 +1100
@@ -12,7 +12,7 @@
     int popularity;
};

-static DEFINE_MUTEX(cache_lock);
+static DEFINE_SPINLOCK(cache_lock);
 static LIST_HEAD(cache);
 static unsigned int cache_num = 0;
 #define MAX_CACHE_SIZE 10
@@ -55,6 +55,7 @@
 int cache_add(int id, const char *name)
 {
     struct object *obj;
+
     unsigned long flags;

     if ((obj = kmalloc(sizeof(*obj), GFP_KERNEL)) == NULL)
         return -ENOMEM;
@@ -63,30 +64,33 @@

```

```

obj->id = id;
obj->popularity = 0;

-    mutex_lock(&cache_lock);
+    spin_lock_irqsave(&cache_lock, flags);
    __cache_add(obj);
-    mutex_unlock(&cache_lock);
+    spin_unlock_irqrestore(&cache_lock, flags);
    return 0;
}

void cache_delete(int id)
{
-    mutex_lock(&cache_lock);
+    unsigned long flags;
+
+    spin_lock_irqsave(&cache_lock, flags);
    __cache_delete(__cache_find(id));
-    mutex_unlock(&cache_lock);
+    spin_unlock_irqrestore(&cache_lock, flags);
}

int cache_find(int id, char *name)
{
    struct object *obj;
    int ret = -ENOENT;
+    unsigned long flags;

-    mutex_lock(&cache_lock);
+    spin_lock_irqsave(&cache_lock, flags);
    obj = __cache_find(id);
    if (obj) {
        ret = 0;
        strcpy(name, obj->name);
    }
-    mutex_unlock(&cache_lock);
+    spin_unlock_irqrestore(&cache_lock, flags);
    return ret;
}

```

Da notare che `spin_lock_irqsave()` disabiliterà le interruzioni se erano attive, altrimenti non farà niente (quando siamo già in un contesto d'interruzione); dunque queste funzioni possono essere chiamate in sicurezza da qualsiasi contesto.

Sfortunatamente, `cache_add()` invoca `kmalloc()` con l'opzione `GFP_KERNEL` che è permessa solo in contesto utente. Ho supposto che `cache_add()` venga chiamata dal contesto utente, altrimenti questa opzione deve diventare un parametro di `cache_add()`.

## Esporre gli oggetti al di fuori del file

Se i vostri oggetti contengono più informazioni, potrebbe non essere sufficiente copiare i dati avanti e indietro: per esempio, altre parti del codice potrebbero avere un puntatore a questi oggetti piuttosto che cercarli ogni volta. Questo introduce due problemi.

Il primo problema è che utilizziamo `cache_lock` per proteggere gli oggetti: dobbiamo renderlo dinamico così che il resto del codice possa usarlo. Questo rende la sincronizzazione più complicata dato che non avviene più in un unico posto.

Il secondo problema è il problema del ciclo di vita: se un'altra struttura mantiene un puntatore ad un oggetto, presumibilmente si aspetta che questo puntatore rimanga valido. Sfortunatamente, questo è garantito solo mentre si trattiene il *lock*, altrimenti qualcuno potrebbe chiamare `cache_delete()` o peggio, aggiungere un oggetto che riutilizza lo stesso indirizzo.

Dato che c'è un solo *lock*, non potete trattenerlo a vita: altrimenti nessun altro potrà eseguire il proprio lavoro.

La soluzione a questo problema è l'uso di un contatore di riferimenti: chiunque punti ad un oggetto deve incrementare il contatore, e decrementarlo quando il puntatore non viene più usato. Quando il contatore raggiunge lo zero significa che non è più usato e l'oggetto può essere rimosso.

Ecco il codice:

```
--- cache.c.interrupt 2003-12-09 14:25:43.000000000 +1100
+++ cache.c.refcnt 2003-12-09 14:33:05.000000000 +1100
@@ -7,6 +7,7 @@
 struct object
 {
     struct list_head list;
+    unsigned int refcnt;
     int id;
     char name[32];
     int popularity;
@@ -17,6 +18,35 @@
     static unsigned int cache_num = 0;
 #define MAX_CACHE_SIZE 10

+static void __object_put(struct object *obj)
+{
+    if (--obj->refcnt == 0)
+        kfree(obj);
+}
+
+static void __object_get(struct object *obj)
+{
+    obj->refcnt++;
+}
+
+void object_put(struct object *obj)
+{
+    unsigned long flags;
```

```

+
+      spin_lock_irqsave(&cache_lock, flags);
+      __object_put(obj);
+      spin_unlock_irqrestore(&cache_lock, flags);
+}
+
+void object_get(struct object *obj)
+{
+    unsigned long flags;
+
+    spin_lock_irqsave(&cache_lock, flags);
+    __object_get(obj);
+    spin_unlock_irqrestore(&cache_lock, flags);
+}
+
/* Must be holding cache_lock */
static struct object *__cache_find(int id)
{
@@ -35,6 +65,7 @@
{
    BUG_ON(!obj);
    list_del(&obj->list);
+    __object_put(obj);
    cache_num--;
}

@@ -63,6 +94,7 @@
    strscpy(obj->name, name, sizeof(obj->name));
    obj->id = id;
    obj->popularity = 0;
+    obj->refcnt = 1; /* The cache holds a reference */

    spin_lock_irqsave(&cache_lock, flags);
    __cache_add(obj);
@@ -79,18 +111,15 @@
    spin_unlock_irqrestore(&cache_lock, flags);
}

-int cache_find(int id, char *name)
+struct object *cache_find(int id)
{
    struct object *obj;
-    int ret = -ENOENT;
    unsigned long flags;

    spin_lock_irqsave(&cache_lock, flags);
    obj = __cache_find(id);
-    if (obj) {
-        ret = 0;
-        strcpy(name, obj->name);
-
```

```

-        }
+        if (obj)
+            __object_get(obj);
+            spin_unlock_irqrestore(&cache_lock, flags);
-        return ret;
+        return obj;
}

```

Abbiamo incapsulato il contatore di riferimenti nelle tipiche funzioni di ‘get’ e ‘put’. Ora possiamo ritornare l’oggetto da cache\_find() col vantaggio che l’utente può dormire trattenendo l’oggetto (per esempio, copy\_to\_user() per copiare il nome verso lo spazio utente).

Un altro punto da notare è che ho detto che il contatore dovrebbe incrementarsi per ogni puntatore ad un oggetto: quindi il contatore di riferimenti è 1 quando l’oggetto viene inserito nella memoria. In altre versione il framework non trattiene un riferimento per se, ma diventa più complicato.

## Usare operazioni atomiche per il contatore di riferimenti

In sostanza, `atomic_t` viene usato come contatore di riferimenti. Ci sono un certo numero di operazioni atomiche definite in `include/asm/atomic.h`: queste sono garantite come atomiche su qualsiasi processore del sistema, quindi non sono necessari i *lock*. In questo caso è più semplice rispetto all’uso degli spinlock, benché l’uso degli spinlock sia più elegante per casi non banali. Le funzioni `atomic_inc()` e `atomic_dec_and_test()` vengono usate al posto dei tipici operatori di incremento e decremento, e i *lock* non sono più necessari per proteggere il contatore stesso.

```

--- cache.c.refcnt 2003-12-09 15:00:35.000000000 +1100
+++ cache.c.refcnt-atomic 2003-12-11 15:49:42.000000000 +1100
@@ -7,7 +7,7 @@
 struct object
 {
     struct list_head list;
-    unsigned int refcnt;
+    atomic_t refcnt;
     int id;
     char name[32];
     int popularity;
@@ -18,33 +18,15 @@
     static unsigned int cache_num = 0;
 #define MAX_CACHE_SIZE 10

-static void __object_put(struct object *obj)
-{ 
-    if (--obj->refcnt == 0)
-        kfree(obj);
-}
-
-static void __object_get(struct object *obj)
-{ 

```

```

-     obj->refcnt++;
- }

-
void object_put(struct object *obj)
{
    unsigned long flags;
-
    spin_lock_irqsave(&cache_lock, flags);
    __object_put(obj);
    spin_unlock_irqrestore(&cache_lock, flags);
+    if (atomic_dec_and_test(&obj->refcnt))
        kfree(obj);
}

void object_get(struct object *obj)
{
    unsigned long flags;
-
    spin_lock_irqsave(&cache_lock, flags);
    __object_get(obj);
    spin_unlock_irqrestore(&cache_lock, flags);
+    atomic_inc(&obj->refcnt);
}

/* Must be holding cache_lock */
@@ -65,7 +47,7 @@
{
    BUG_ON(!obj);
    list_del(&obj->list);
-    __object_put(obj);
+    object_put(obj);
    cache_num--;
}

@@ -94,7 +76,7 @@
    strscpy(obj->name, name, sizeof(obj->name));
    obj->id = id;
    obj->popularity = 0;
-    obj->refcnt = 1; /* The cache holds a reference */
+    atomic_set(&obj->refcnt, 1); /* The cache holds a reference */

    spin_lock_irqsave(&cache_lock, flags);
    __cache_add(obj);
@@ -119,7 +101,7 @@
    spin_lock_irqsave(&cache_lock, flags);
    obj = __cache_find(id);
    if (obj)
-        __object_get(obj);
+        object_get(obj);
    spin_unlock_irqrestore(&cache_lock, flags);
}

```

```
    return obj;
}
```

## Proteggere l'oggetto stesso

In questo esempio, assumiamo che gli oggetti (ad eccezione del contatore di riferimenti) non cambino mai dopo la loro creazione. Se vogliamo permettere al nome di cambiare abbiamo tre possibilità:

- Si può togliere static da `cache_lock` e dire agli utenti che devono trattenere il *lock* prima di modificare il nome di un oggetto.
- Si può fornire una funzione `cache_obj_rename()` che prende il *lock* e cambia il nome per conto del chiamante; si dirà poi agli utenti di usare questa funzione.
- Si può decidere che `cache_lock` protegge solo la memoria stessa, ed un altro *lock* è necessario per la protezione del nome.

Teoricamente, possiamo avere un *lock* per ogni campo e per ogni oggetto. In pratica, le varianti più comuni sono:

- un *lock* che protegge l'infrastruttura (la lista cache di questo esempio) e gli oggetti. Questo è quello che abbiamo fatto finora.
- un *lock* che protegge l'infrastruttura (inclusi i puntatori alla lista negli oggetti), e un *lock* nell'oggetto per proteggere il resto dell'oggetto stesso.
- *lock* multipli per proteggere l'infrastruttura (per esempio un *lock* per ogni lista), possibilmente con un *lock* per oggetto.

Qui di seguito un'implementazione con “un lock per oggetto”:

```
--- cache.c.refcnt-atomic 2003-12-11 15:50:54.000000000 +1100
+++ cache.c.perobjectlock 2003-12-11 17:15:03.000000000 +1100
@@ -6,11 +6,17 @@
 
 struct object
 {
+   /* These two protected by cache_lock. */
+   struct list_head list;
+   int popularity;
+
+   atomic_t refcnt;
+
+   /* Doesn't change once created. */
+   int id;
+
+   spinlock_t lock; /* Protects the name */
+   char name[32];
-   int popularity;
};

static DEFINE_SPINLOCK(cache_lock);
```

```
@@ -77,6 +84,7 @@
        obj->id = id;
        obj->popularity = 0;
        atomic_set(&obj->refcnt, 1); /* The cache holds a reference */
+
        spin_lock_init(&obj->lock);

        spin_lock_irqsave(&cache_lock, flags);
        __cache_add(obj);
```

Da notare che ho deciso che il contatore di popolarità dovesse essere protetto da `cache_lock` piuttosto che dal `lock` dell'oggetto; questo perché è logicamente parte dell'infrastruttura (come `struct list_head` nell'oggetto). In questo modo, in `__cache_add()`, non ho bisogno di trattenere il `lock` di ogni oggetto mentre si cerca il meno popolare.

Ho anche deciso che il campo `id` è immutabile, quindi non ho bisogno di trattenere il `lock` dell'oggetto quando si usa `__cache_find()` per leggere questo campo; il `lock` dell'oggetto è usato solo dal chiamante che vuole leggere o scrivere il campo `name`.

Inoltre, da notare che ho aggiunto un commento che descrive i dati che sono protetti dal `lock`. Questo è estremamente importante in quanto descrive il comportamento del codice, che altrimenti sarebbe di difficile comprensione leggendo solamente il codice. E come dice Alan Cox: "Lock data, not code".

## Problemi comuni

### Stallo: semplice ed avanzato

Esiste un tipo di baco dove un pezzo di codice tenta di trattenere uno spinlock due volte: questo rimarrà in attesa attiva per sempre aspettando che il `lock` venga rilasciato (in Linux spinlocks, rwlocks e mutex non sono ricorsivi). Questo è facile da diagnosticare: non è uno di quei problemi che ti tengono sveglio 5 notti a parlare da solo.

Un caso un pochino più complesso; immaginate d'avere una spazio condiviso fra un softirq ed il contesto utente. Se usate `spin_lock()` per proteggerlo, il contesto utente potrebbe essere interrotto da un softirq mentre trattiene il lock, da qui il softirq rimarrà in attesa attiva provando ad acquisire il `lock` già trattenuto nel contesto utente.

Questi casi sono chiamati stalli (*deadlock*), e come mostrato qui sopra, può succedere anche con un solo processore (Ma non sui sistemi monoprocessoressi perché gli spinlock spariscano quando il kernel è compilato con `CONFIG_SMP=n`. Nonostante ciò, nel secondo caso avrete comunque una corruzione dei dati).

Questi casi sono facili da diagnosticare; sui sistemi multi-processore il supervisione (*watchdog*) o l'opzione di compilazione `DEBUG_SPINLOCK` (`include/linux/spinlock.h`) permettono di scovare immediatamente quando succedono.

Esiste un caso più complesso che è conosciuto come l'abbraccio della morte; questo coinvolge due o più `lock`. Diciamo che avete un vettore di hash in cui ogni elemento è uno spinlock a cui è associata una lista di elementi con lo stesso hash. In un gestore di interruzioni software, dovete modificare un oggetto e spostarlo su un altro hash; quindi dovete trattenere lo spinlock del vecchio hash e di quello nuovo, quindi rimuovere l'oggetto dal vecchio ed inserirlo nel nuovo.

Qui abbiamo due problemi. Primo, se il vostro codice prova a spostare un oggetto all'interno della stessa lista, otterrete uno stalllo visto che tenterà di trattenere lo stesso *lock* due volte. Secondo, se la stessa interruzione software su un altro processore sta tentando di spostare un altro oggetto nella direzione opposta, potrebbe accadere quanto segue:

CPU 1	CPU 2
Trattiene <i>lock A</i> -> OK	Trattiene <i>lock B</i> -> OK
Trattiene <i>lock B</i> -> attesa	Trattiene <i>lock A</i> -> attesa

Table: Conseguenze

Entrambe i processori rimarranno in attesa attiva sul *lock* per sempre, aspettando che l'altro lo rilasci. Sembra e puzza come un blocco totale.

### Prevenire gli stalli

I libri di testo vi diranno che se trattenete i *lock* sempre nello stesso ordine non avrete mai un simile stalllo. La pratica vi dirà che questo approccio non funziona all'ingrandirsi del sistema: quando creo un nuovo *lock* non ne capisco abbastanza del kernel per dire in quale dei 5000 *lock* si incastrerà.

I *lock* migliori sono quelli encapsulati: non vengono esposti nei file di intestazione, e non vengono mai trattenuti fuori dallo stesso file. Potete rileggere questo codice e vedere che non ci sarà mai uno stalllo perché non tenterà mai di trattenere un altro *lock* quando lo ha già. Le persone che usano il vostro codice non devono nemmeno sapere che voi state usando dei *lock*.

Un classico problema deriva dall'uso di *callback* e di *hook*: se li chiamate mentre trattenete un *lock*, rischiate uno stalllo o un abbraccio della morte (chi lo sa cosa farà una *callback*?).

### Ossessiva prevenzione degli stalli

Gli stalli sono un problema, ma non così terribile come la corruzione dei dati. Un pezzo di codice trattiene un *lock* di lettura, cerca in una lista, fallisce nel trovare quello che vuole, quindi rilascia il *lock* di lettura, trattiene un *lock* di scrittura ed inserisce un oggetto; questo genere di codice presenta una corsa critica.

Se non riuscite a capire il perché, per favore state alla larga dal mio codice.

### corsa fra temporizzatori: un passatempo del kernel

I temporizzatori potrebbero avere dei problemi con le corse critiche. Considerate una collezione di oggetti (liste, hash, eccetera) dove ogni oggetto ha un temporizzatore che sta per distruggerlo.

Se volete eliminare l'intera collezione (diciamo quando rimuovete un modulo), potreste fare come segue:

```
/* THIS CODE BAD BAD BAD BAD: IF IT WAS ANY WORSE IT WOULD USE
   HUNGARIAN NOTATION */
spin_lock_bh(&list_lock);
```

```

while (list) {
    struct foo *next = list->next;
    del_timer(&list->timer);
    kfree(list);
    list = next;
}

spin_unlock_bh(&list_lock);

```

Primo o poi, questo esploderà su un sistema multiprocessore perché un temporizzatore potrebbe essere già partito prima di `spin_lock_bh()`, e prenderà il *lock* solo dopo `spin_unlock_bh()`, e cercherà di eliminare il suo oggetto (che però è già stato eliminato).

Questo può essere evitato controllando il valore di ritorno di `del_timer()`: se ritorna 1, il temporizzatore è stato già rimosso. Se 0, significa (in questo caso) che il temporizzatore è in esecuzione, quindi possiamo fare come segue:

```

retry:
    spin_lock_bh(&list_lock);

    while (list) {
        struct foo *next = list->next;
        if (!del_timer(&list->timer)) {
            /* Give timer a chance to delete this */
            spin_unlock_bh(&list_lock);
            goto retry;
        }
        kfree(list);
        list = next;
    }

    spin_unlock_bh(&list_lock);

```

Un altro problema è l'eliminazione dei temporizzatori che si riavviano da soli (chiamando `add_timer()` alla fine della loro esecuzione). Dato che questo è un problema abbastanza comune con una propensione alle corse critiche, dovreste usare `del_timer_sync()` (`include/linux/timer.h`) per gestire questo caso. Questa ritorna il numero di volte che il temporizzatore è stato interrotto prima che fosse in grado di fermarlo senza che si riavvisasse.

## Velocità della sincronizzazione

Ci sono tre cose importanti da tenere in considerazione quando si valuta la velocità d'esecuzione di un pezzo di codice che necessita di sincronizzazione. La prima è la concorrenza: quante cose rimangono in attesa mentre qualcuno trattiene un *lock*. La seconda è il tempo necessario per acquisire (senza contese) e rilasciare un *lock*. La terza è di usare meno *lock* o di più furbi. Immagino che i *lock* vengano usati regolarmente, altrimenti, non sareste interessati all'efficienza.

La concorrenza dipende da quanto a lungo un *lock* è trattenuto: dovreste trattenere un *lock* solo il tempo minimo necessario ma non un istante in più. Nella memoria dell'esempio precedente,

creiamo gli oggetti senza trattenere il *lock*, poi acquisiamo il *lock* quando siamo pronti per inserirlo nella lista.

Il tempo di acquisizione di un *lock* dipende da quanto danno fa l'operazione sulla *pipeline* (ovvero stalli della *pipeline*) e quant'è probabile che il processore corrente sia stato anche l'ultimo ad acquisire il *lock* (in pratica, il *lock* è nella memoria cache del processore corrente?): su sistemi multi-processore questa probabilità precipita rapidamente. Consideriamo un processore Intel Pentium III a 700Mhz: questo esegue un'istruzione in 0.7ns, un incremento atomico richiede 58ns, acquisire un *lock* che è nella memoria cache del processore richiede 160ns, e un trasferimento dalla memoria cache di un altro processore richiede altri 170/360ns (Leggetevi l'articolo di Paul McKenney's [Linux Journal RCU article](#)).

Questi due obiettivi sono in conflitto: trattenere un *lock* per il minor tempo possibile potrebbe richiedere la divisione in più *lock* per diverse parti (come nel nostro ultimo esempio con un *lock* per ogni oggetto), ma questo aumenta il numero di acquisizioni di *lock*, ed il risultato spesso è che tutto è più lento che con un singolo *lock*. Questo è un altro argomento in favore della semplicità quando si parla di sincronizzazione.

Il terzo punto è discusso di seguito: ci sono alcune tecniche per ridurre il numero di sincronizzazioni che devono essere fatte.

### Read/Write Lock Variants

Sia gli spinlock che i mutex hanno una variante per la lettura/scrittura (read/write): `rwlock_t` e `struct rw_semaphore`. Queste dividono gli utenti in due categorie: i lettori e gli scrittori. Se state solo leggendo i dati, potete acquisire il *lock* di lettura, ma per scrivere avrete bisogno del *lock* di scrittura. Molti possono trattenere il *lock* di lettura, ma solo uno scrittore alla volta può trattenere quello di scrittura.

Se il vostro codice si divide chiaramente in codice per lettori e codice per scrittori (come nel nostro esempio), e il *lock* dei lettori viene trattenuto per molto tempo, allora l'uso di questo tipo di *lock* può aiutare. Questi sono leggermente più lenti rispetto alla loro versione normale, quindi nella pratica l'uso di `rwlock_t` non ne vale la pena.

### Evitare i *lock*: Read Copy Update

Esiste un metodo di sincronizzazione per letture e scritture detto Read Copy Update. Con l'uso della tecnica RCU, i lettori possono scordarsi completamente di trattenere i *lock*; dato che nel nostro esempio ci aspettiamo d'avere più lettore che scrittori (altrimenti questa memoria sarebbe uno spreco) possiamo dire che questo meccanismo permette un'ottimizzazione.

Come facciamo a sbarazzarci dei *lock* di lettura? Sbarazzarsi dei *lock* di lettura significa che uno scrittore potrebbe cambiare la lista sotto al naso dei lettori. Questo è abbastanza semplice: possiamo leggere una lista concatenata se lo scrittore aggiunge elementi alla fine e con certe precauzioni. Per esempio, aggiungendo `new` ad una lista concatenata chiamata `list`:

```
new->next = list->next;
wmb();
list->next = new;
```

La funzione `wmb()` è una barriera di sincronizzazione delle scritture. Questa garantisce che la prima operazione (impostare l'elemento `next` del nuovo elemento) venga completata e vista

da tutti i processori prima che venga eseguita la seconda operazione (che sarebbe quella di mettere il nuovo elemento nella lista). Questo è importante perché i moderni compilatori ed i moderni processori possono, entrambe, riordinare le istruzioni se non vengono istruiti altrimenti: vogliamo che i lettori non vedano completamente il nuovo elemento; oppure che lo vedano correttamente e quindi il puntatore `next` deve puntare al resto della lista.

Fortunatamente, c'è una funzione che fa questa operazione sulle liste `struct list_head`: `list_add_rcu()` (`include/linux/list.h`).

Rimuovere un elemento dalla lista è anche più facile: sostituiamo il puntatore al vecchio elemento con quello del suo successore, e i lettori vedranno l'elemento o lo salteranno.

```
list->next = old->next;
```

La funzione `list_del_rcu()` (`include/linux/list.h`) fa esattamente questo (la versione normale corrompe il vecchio oggetto, e non vogliamo che accada).

Anche i lettori devono stare attenti: alcuni processori potrebbero leggere attraverso il puntatore `next` il contenuto dell'elemento successivo troppo presto, ma non accorgersi che il contenuto caricato è sbagliato quando il puntatore `next` viene modificato alla loro spalle. Ancora una volta c'è una funzione che viene in vostro aiuto `list_for_each_entry_rcu()` (`include/linux/list.h`). Ovviamente, gli scrittori possono usare `list_for_each_entry()` dato che non ci possono essere due scrittori in contemporanea.

Il nostro ultimo dilemma è il seguente: quando possiamo realmente distruggere l'elemento rimosso? Ricordate, un lettore potrebbe aver avuto accesso a questo elemento proprio ora: se eliminiamo questo elemento ed il puntatore `next` cambia, il lettore salterà direttamente nella spazzatura e scoppiera. Dobbiamo aspettare finché tutti i lettori che stanno attraversando la lista abbiano finito. Utilizziamo `call_rcu()` per registrare una funzione di richiamo che distrugga l'oggetto quando tutti i lettori correnti hanno terminato. In alternativa, potrebbe essere usata la funzione `synchronize_rcu()` che blocca l'esecuzione finché tutti i lettori non terminano di ispezionare la lista.

Ma come fa l'RCU a sapere quando i lettori sono finiti? Il meccanismo è il seguente: innanzi tutto i lettori accedono alla lista solo fra la coppia `rcu_read_lock()`/`rcu_read_unlock()` che disabilita la prelazione così che i lettori non vengano sospesi mentre stanno leggendo la lista.

Poi, l'RCU aspetta finché tutti i processori non abbiano dormito almeno una volta; a questo punto, dato che i lettori non possono dormire, possiamo dedurre che un qualsiasi lettore che abbia consultato la lista durante la rimozione abbia già terminato, quindi la `callback` viene eseguita. Il vero codice RCU è un po' più ottimizzato di così, ma questa è l'idea di fondo.

```
-- cache.c.perobjectlock 2003-12-11 17:15:03.000000000 +1100
+++ cache.c.rcupdate    2003-12-11 17:55:14.000000000 +1100
@@ -1,15 +1,18 @@
 #include <linux/list.h>
 #include <linux/slab.h>
 #include <linux/string.h>
+#include <linux/rcupdate.h>
 #include <linux/mutex.h>
 #include <asm/errno.h>

 struct object
 {
-     /* These two protected by cache_lock. */

```

```

+     /* This is protected by RCU */
+     struct list_head list;
+     int popularity;

+     struct rcu_head rcu;
+
+     atomic_t refcnt;

     /* Doesn't change once created. */
@@ -40,7 +43,7 @@
{
    struct object *i;

-    list_for_each_entry(i, &cache, list) {
+    list_for_each_entry_rcu(i, &cache, list) {
        if (i->id == id) {
            i->popularity++;
            return i;
@@ -49,19 +52,25 @@
        return NULL;
    }

+/* Final discard done once we know no readers are looking. */
+static void cache_delete_rcu(void *arg)
+{
+    object_put(arg);
+}
+
/* Must be holding cache_lock */
static void __cache_delete(struct object *obj)
{
    BUG_ON(!obj);
-    list_del(&obj->list);
-    object_put(obj);
+    list_del_rcu(&obj->list);
    cache_num--;
+    call_rcu(&obj->rcu, cache_delete_rcu);
}

/* Must be holding cache_lock */
static void __cache_add(struct object *obj)
{
-    list_add(&obj->list, &cache);
+    list_add_rcu(&obj->list, &cache);
    if (++cache_num > MAX_CACHE_SIZE) {
        struct object *i, *outcast = NULL;
        list_for_each_entry(i, &cache, list) {
@@ -104,12 +114,11 @@
struct object *cache_find(int id)
{

```

```

- struct object *obj;
- unsigned long flags;

- spin_lock_irqsave(&cache_lock, flags);
+ rcu_read_lock();
obj = __cache_find(id);
if (obj)
    object_get(obj);
- spin_unlock_irqrestore(&cache_lock, flags);
+ rcu_read_unlock();
return obj;
}

```

Da notare che i lettori modificano il campo `popularity` nella funzione `__cache_find()`, e ora non trattiene alcun *lock*. Una soluzione potrebbe essere quella di rendere la variabile `atomic_t`, ma per l'uso che ne abbiamo fatto qui, non ci interessano queste corse critiche perché un risultato approssimativo è comunque accettabile, quindi non l'ho cambiato.

Il risultato è che la funzione `cache_find()` non ha bisogno di alcuna sincronizzazione con le altre funzioni, quindi è veloce su un sistema multi-processore tanto quanto lo sarebbe su un sistema mono-processore.

Esiste un'ulteriore ottimizzazione possibile: vi ricordate il codice originale della nostra memoria dove non c'erano contatori di riferimenti e il chiamante semplicemente tratteneva il *lock* prima di accedere ad un oggetto? Questo è ancora possibile: se trattenete un *lock* nessuno potrà cancellare l'oggetto, quindi non avete bisogno di incrementare e decrementare il contatore di riferimenti.

Ora, dato che il '*lock* di lettura' di un RCU non fa altro che disabilitare la prelazione, un chiamante che ha sempre la prelazione disabilitata fra le chiamate `cache_find()` e `object_put()` non necessita di incrementare e decrementare il contatore di riferimenti. Potremmo esporre la funzione `__cache_find()` dichiarandola non-static, e quel chiamante potrebbe usare direttamente questa funzione.

Il beneficio qui sta nel fatto che il contatore di riferimenti no viene scritto: l'oggetto non viene alterato in alcun modo e quindi diventa molto più veloce su sistemi multi-processore grazie alla loro memoria cache.

## Dati per processore

Un'altra tecnica comunemente usata per evitare la sincronizzazione è quella di duplicare le informazioni per ogni processore. Per esempio, se volete avere un contatore di qualcosa, potreste utilizzare uno spinlock ed un singolo contatore. Facile e pulito.

Se questo dovesse essere troppo lento (solitamente non lo è, ma se avete dimostrato che lo è devvero), potreste usare un contatore per ogni processore e quindi non sarebbe più necessaria la mutua esclusione. Vedere `DEFINE_PER_CPU()`, `get_cpu_var()` e `put_cpu_var()` (`include/linux/percpu.h`).

Il tipo di dato `local_t`, la funzione `cpu_local_inc()` e tutte le altre funzioni associate, sono di particolare utilità per semplici contatori per-processore; su alcune architetture sono anche più efficienti (`include/asm/local.h`).

Da notare che non esiste un modo facile ed affidabile per ottenere il valore di un simile contatore senza introdurre altri *lock*. In alcuni casi questo non è un problema.

### Dati che sono usati prevalentemente dai gestori d'interruzioni

Se i dati vengono utilizzati sempre dallo stesso gestore d'interruzioni, allora i *lock* non vi servono per niente: il kernel già vi garantisce che il gestore d'interruzione non verrà eseguito in contemporanea su diversi processori.

Manfred Spraul fa notare che potreste comunque comportarvi così anche se i dati vengono occasionalmente utilizzati da un contesto utente o da un'interruzione software. Il gestore d'interruzione non utilizza alcun *lock*, e tutti gli altri accessi verranno fatti così:

```
spin_lock(&lock);
disable_irq(irq);
...
enable_irq(irq);
spin_unlock(&lock);
```

La funzione `disable_irq()` impedisce al gestore d'interruzioni d'essere eseguito (e aspetta che finisca nel caso fosse in esecuzione su un altro processore). Lo spinlock, invece, previene accessi simultanei. Naturalmente, questo è più lento della semplice chiamata `spin_lock_irq()`, quindi ha senso solo se questo genere di accesso è estremamente raro.

### Quali funzioni possono essere chiamate in modo sicuro dalle interruzioni?

Molte funzioni del kernel dormono (in sostanza, chiamano `schedule()`) direttamente od indirettamente: non potete chiamarle se trattenere uno spinlock o avete la prelazione disabilitata, mai. Questo significa che dovete necessariamente essere nel contesto utente: chiamarle da un contesto d'interruzione è illegale.

### Alcune funzioni che dormono

Le più comuni sono elencate qui di seguito, ma solitamente dovete leggere il codice per scoprire se altre chiamate sono sicure. Se chiunque altro le chiama dorme, allora dovreste poter dormire anche voi. In particolar modo, le funzioni di registrazione e deregistrazione solitamente si aspettano d'essere chiamante da un contesto utente e quindi che possono dormire.

- Accessi allo spazio utente:
  - `copy_from_user()`
  - `copy_to_user()`
  - `get_user()`
  - `put_user()`
- `kmalloc(GFP_KERNEL) <kmalloc>`
- `mutex_lock_interruptible()` and `mutex_lock()`

C'è anche `mutex_trylock()` che però non dorme. Comunque, non deve essere usata in un contesto d'interruzione dato che la sua implementazione non è sicura in quel contesto. Anche `mutex_unlock()` non dorme mai. Non può comunque essere usata in un contesto d'interruzione perché un mutex deve essere rilasciato dallo stesso processo che l'ha acquisito.

## Alcune funzioni che non dormono

Alcune funzioni possono essere chiamate tranquillamente da qualsiasi contesto, o trattenendo un qualsiasi *lock*.

- `printk()`
- `kfree()`
- `add_timer()` e `del_timer()`

## Riferimento per l'API dei Mutex

### **mutex\_init**

`mutex_init (mutex)`

initialize the mutex

#### Parameters

**mutex** the mutex to be initialized

#### Description

Initialize the mutex to unlocked state.

It is not allowed to initialize an already locked mutex.

`bool mutex_is_locked(struct mutex *lock)`  
is the mutex locked

#### Parameters

**struct mutex \*lock** the mutex to be queried

#### Description

Returns true if the mutex is locked, false if unlocked.

`void mutex_lock(struct mutex *lock)`  
acquire the mutex

#### Parameters

**struct mutex \*lock** the mutex to be acquired

#### Description

Lock the mutex exclusively for this task. If the mutex is not available right now, it will sleep until it can get it.

The mutex must later on be released by the same task that acquired it. Recursive locking is not allowed. The task may not exit without first unlocking the mutex. Also, kernel memory

where the mutex resides must not be freed with the mutex still locked. The mutex must first be initialized (or statically defined) before it can be locked. memset()-ing the mutex to 0 is not allowed.

(The CONFIG\_DEBUG\_MUTEXES .config option turns on debugging checks that will enforce the restrictions and will also do deadlock debugging)

This function is similar to (but not equivalent to) down().

```
void mutex_unlock(struct mutex *lock)
    release the mutex
```

### Parameters

**struct mutex \*lock** the mutex to be released

### Description

Unlock a mutex that has been locked by this task previously.

This function must not be used in interrupt context. Unlocking of a not locked mutex is not allowed.

This function is similar to (but not equivalent to) up().

```
void ww_mutex_unlock(struct ww_mutex *lock)
    release the w/w mutex
```

### Parameters

**struct ww\_mutex \*lock** the mutex to be released

### Description

Unlock a mutex that has been locked by this task previously with any of the ww\_mutex\_lock\* functions (with or without an acquire context). It is forbidden to release the locks after releasing the acquire context.

This function must not be used in interrupt context. Unlocking of a unlocked mutex is not allowed.

```
int ww_mutex_trylock(struct ww_mutex *ww, struct ww_acquire_ctx *ww_ctx)
    tries to acquire the w/w mutex with optional acquire context
```

### Parameters

**struct ww\_mutex \*ww** mutex to lock

**struct ww\_acquire\_ctx \*ww\_ctx** optional w/w acquire context

### Description

Trylocks a mutex with the optional acquire context; no deadlock detection is possible. Returns 1 if the mutex has been acquired successfully, 0 otherwise.

Unlike ww\_mutex\_lock, no deadlock handling is performed. However, if a **ctx** is specified, -EALREADY handling may happen in calls to ww\_mutex\_trylock.

A mutex acquired with this function must be released with ww\_mutex\_unlock.

```
int mutex_lock_interruptible(struct mutex *lock)
    Acquire the mutex, interruptible by signals.
```

### Parameters

**struct mutex \*lock** The mutex to be acquired.

## Description

Lock the mutex like mutex\_lock(). If a signal is delivered while the process is sleeping, this function will return without acquiring the mutex.

## Context

Process context.

## Return

0 if the lock was successfully acquired or -EINTR if a signal arrived.

**int mutex\_lock\_killable(struct mutex \*lock)**

Acquire the mutex, interruptible by fatal signals.

## Parameters

**struct mutex \*lock** The mutex to be acquired.

## Description

Lock the mutex like mutex\_lock(). If a signal which will be fatal to the current process is delivered while the process is sleeping, this function will return without acquiring the mutex.

## Context

Process context.

## Return

0 if the lock was successfully acquired or -EINTR if a fatal signal arrived.

**void mutex\_lock\_io(struct mutex \*lock)**

Acquire the mutex and mark the process as waiting for I/O

## Parameters

**struct mutex \*lock** The mutex to be acquired.

## Description

Lock the mutex like mutex\_lock(). While the task is waiting for this mutex, it will be accounted as being in the IO wait state by the scheduler.

## Context

Process context.

**int mutex\_trylock(struct mutex \*lock)**

try to acquire the mutex, without waiting

## Parameters

**struct mutex \*lock** the mutex to be acquired

## Description

Try to acquire the mutex atomically. Returns 1 if the mutex has been acquired successfully, and 0 on contention.

This function must not be used in interrupt context. The mutex must be released by the same task that acquired it.

### NOTE

this function follows the spin\_trylock() convention, so it is negated from the down\_trylock() return values! Be careful about this when converting semaphore users to mutexes.

```
int atomic_dec_and_mutex_lock(atomic_t *cnt, struct mutex *lock)
    return holding mutex if we dec to 0
```

### Parameters

**atomic\_t \*cnt** the atomic which we are to dec

**struct mutex \*lock** the mutex to return holding if we dec to 0

### Description

return true and hold lock if we dec to 0, return false otherwise

## Riferimento per l'API dei Futex

```
struct futex_hash_bucket *futex_hash(union futex_key *key)
```

Return the hash bucket in the global hash

### Parameters

**union futex\_key \*key** Pointer to the futex key for which the hash is calculated

### Description

We hash on the keys returned from get\_futex\_key (see below) and return the corresponding hash bucket in the global hash.

```
struct hrtimer_sleeper *futex_setup_timer(ktime_t *time, struct hrtimer_sleeper *timeout,
                                             int flags, u64 range_ns)
```

set up the sleeping hrtimer.

### Parameters

**ktime\_t \*time** ptr to the given timeout value

**struct hrtimer\_sleeper \*timeout** the hrtimer\_sleeper structure to be set up

**int flags** futex flags

**u64 range\_ns** optional range in ns

### Return

**Initialized hrtimer\_sleeper structure or NULL if no timeout value given**

```
int get_futex_key(u32 __user *uaddr, bool fshared, union futex_key *key, enum futex_access
                   rw)
```

Get parameters which are the keys for a futex

### Parameters

**u32 \_\_user \*uaddr** virtual address of the futex

**bool fshared** false for a PROCESS\_PRIVATE futex, true for PROCESS\_SHARED

**union futex\_key \*key** address where result is stored.

**enum futex\_access rw** mapping needs to be read/write (values: FUTEX\_READ, FUTEX\_WRITE)

## Return

a negative error code or 0

## Description

The key words are stored in **key** on success.

For shared mappings (when **fshared**), the key is:

(inode->i\_sequence, page->index, offset\_within\_page )

[ also see get\_inode\_sequence\_number() ]

For private mappings (or when **!fshared**), the key is:

( current->mm, address, 0 )

This allows (cross process, where applicable) identification of the futex without keeping the page pinned for the duration of the FUTEX\_WAIT.

lock\_page() might sleep, the caller should not hold a spinlock.

**int fault\_in\_user\_writeable(u32 \_\_user \*uaddr)**

Fault in user address and verify RW access

## Parameters

**u32 \_\_user \*uaddr** pointer to faulting user space address

## Description

Slow path to fixup the fault we just took in the atomic write access to **uaddr**.

We have no generic implementation of a non-destructive write to the user address. We know that we faulted in the atomic pagefault disabled section so we can as well avoid the #PF overhead by calling get\_user\_pages() right away.

**struct futex\_q \*futex\_top\_waiter(struct futex\_hash\_bucket \*hb, union futex\_key \*key)**

Return the highest priority waiter on a futex

## Parameters

**struct futex\_hash\_bucket \*hb** the hash bucket the futex\_q's reside in

**union futex\_key \*key** the futex key (to distinguish it from other futex futex\_q's)

## Description

Must be called with the hb lock held.

**void wait\_for\_owner\_exiting(int ret, struct task\_struct \*exiting)**

Block until the owner has exited

## Parameters

**int ret** owner's current futex lock status

**struct task\_struct \*exiting** Pointer to the exiting task

## Description

Caller must hold a refcount on **exiting**.

```
void __futex_unqueue(struct futex_q *q)
    Remove the futex_q from its futex_hash_bucket
```

### Parameters

**struct futex\_q \*q** The futex\_q to unqueue

### Description

The q->lock\_ptr must not be NULL and must be held by the caller.

```
int futex_unqueue(struct futex_q *q)
    Remove the futex_q from its futex_hash_bucket
```

### Parameters

**struct futex\_q \*q** The futex\_q to unqueue

### Description

The q->lock\_ptr must not be held by the caller. A call to futex\_unqueue() must be paired with exactly one earlier call to futex\_queue().

### Return

- 1 - if the futex\_q was still queued (and we removed unqueued it);
- 0 - if the futex\_q was already removed by the waking thread

```
void futex_exit_recursive(struct task_struct *tsk)
    Set the tasks futex state to FUTEX_STATE_DEAD
```

### Parameters

**struct task\_struct \*tsk** task to set the state on

### Description

Set the futex exit state of the task lockless. The futex waiter code observes that state when a task is exiting and loops until the task has actually finished the futex cleanup. The worst case for this is that the waiter runs through the wait loop until the state becomes visible.

This is called from the recursive fault handling path in make\_task\_dead().

This is best effort. Either the futex exit code has run already or not. If the OWNER\_DIED bit has been set on the futex then the waiter can take it over. If not, the problem is pushed back to user space. If the futex exit code did not run yet, then an already queued waiter might block forever, but there is nothing which can be done about that.

### struct futex\_q

The hashed futex queue entry, one per waiting task

### Definition

```
struct futex_q {
    struct plist_node list;
    struct task_struct *task;
    spinlock_t *lock_ptr;
    union futex_key key;
    struct futex_pi_state *pi_state;
    struct rt_mutex_waiter *rt_waiter;
    union futex_key *requeue_pi_key;
```

```

u32 bitset;
atomic_t requeue_state;
#ifndef CONFIG_PREEMPT_RT;
    struct rcuwait requeue_wait;
#endif;
};

```

## Members

**list** priority-sorted list of tasks waiting on this futex

**task** the task waiting on the futex

**lock\_ptr** the hash bucket lock

**key** the key the futex is hashed on

**pi\_state** optional priority inheritance state

**rt\_waiter** rt\_waiter storage for use with requeue\_pi

**requeue\_pi\_key** the requeue\_pi target futex key

**bitset** bitset for the optional bitmasked wakeup

**requeue\_state** State field for futex\_requeue\_pi()

**requeue\_wait** RCU wait for futex\_requeue\_pi() (RT only)

## Description

We use this hashed waitqueue, instead of a normal wait\_queue\_entry\_t, so we can wake only the relevant ones (hashed queues may be shared).

A futex\_q has a woken state, just like tasks have TASK\_RUNNING. It is considered woken when plist\_node\_empty(q->list) || q->lock\_ptr == 0. The order of wakeup is always to make the first condition true, then the second.

PI futexes are typically woken before they are removed from the hash list via the rt\_mutex code. See futex\_unqueue\_pi().

int **futex\_match**(union futex\_key \*key1, union futex\_key \*key2)

Check whether two futex keys are equal

## Parameters

**union futex\_key \*key1** Pointer to key1

**union futex\_key \*key2** Pointer to key2

## Description

Return 1 if two futex\_keys are equal, 0 otherwise.

void **futex\_enqueue**(struct *futex\_q* \*q, struct futex\_hash\_bucket \*hb)

Enqueue the futex\_q on the futex\_hash\_bucket

## Parameters

**struct futex\_q \*q** The futex\_q to enqueue

**struct futex\_hash\_bucket \*hb** The destination hash bucket

### Description

The hb->lock must be held by the caller, and is released here. A call to futex\_queue() is typically paired with exactly one call to futex\_unqueue(). The exceptions involve the PI related operations, which may use futex\_unqueue\_pi() or nothing if the unqueue is done as part of the wake process and the unqueue state is implicit in the state of woken task (see futex\_wait\_requeue\_pi() for an example).

### struct **futex\_vector**

Auxiliary struct for futex\_waitv()

### Definition

```
struct futex_vector {  
    struct futex_waitv w;  
    struct futex_q q;  
};
```

### Members

**w** Userspace provided data

**q** Kernel side data

### Description

Struct used to build an array with all data need for futex\_waitv()

```
int futex_lock_pi_atomic(u32 __user *uaddr, struct futex_hash_bucket *hb, union futex_key  
    *key, struct futex_pi_state **ps, struct task_struct *task, struct  
    task_struct **exiting, int set_waiters)
```

Atomic work required to acquire a pi aware futex

### Parameters

**u32 \_\_user \*uaddr** the pi futex user address

**struct futex\_hash\_bucket \*hb** the pi futex hash bucket

**union futex\_key \*key** the futex key associated with uaddr and hb

**struct futex\_pi\_state \*\*ps** the pi\_state pointer where we store the result of the lookup

**struct task\_struct \*task** the task to perform the atomic lock work for. This will be “current” except in the case of requeue pi.

**struct task\_struct \*\*exiting** Pointer to store the task pointer of the owner task which is in the middle of exiting

**int set\_waiters** force setting the FUTEX\_WAITERS bit (1) or not (0)

### Return

- 0 - ready to wait;
- 1 - acquired the lock;
- <0 - error

### Description

The hb->lock must be held by the caller.

**exiting** is only set when the return value is -EBUSY. If so, this holds a refcount on the exiting task on return and the caller needs to drop it after waiting for the exit to complete.

```
int fixup_pi_owner(u32 __user *uaddr, struct futex_q *q, int locked)
    Post lock pi_state and corner case management
```

## Parameters

**u32 \_\_user \*uaddr** user address of the futex  
**struct futex\_q \*q** futex\_q (contains pi\_state and access to the rt\_mutex)  
**int locked** if the attempt to take the rt\_mutex succeeded (1) or not (0)

## Description

After attempting to lock an rt\_mutex, this function is called to cleanup the pi\_state owner as well as handle race conditions that may allow us to acquire the lock. Must be called with the hb lock held.

## Return

- 1 - success, lock taken;
- 0 - success, lock not taken;
- <0 - on error (-EFAULT)

```
void requeue_futex(struct futex_q *q, struct futex_hash_bucket *hb1, struct
                    futex_hash_bucket *hb2, union futex_key *key2)
    Requeue a futex_q from one hb to another
```

## Parameters

**struct futex\_q \*q** the futex\_q to requeue  
**struct futex\_hash\_bucket \*hb1** the source hash\_bucket  
**struct futex\_hash\_bucket \*hb2** the target hash\_bucket  
**union futex\_key \*key2** the new key for the requeued futex\_q

```
void requeue_pi_wake_futex(struct futex_q *q, union futex_key *key, struct
                            futex_hash_bucket *hb)
    Wake a task that acquired the lock during requeue
```

## Parameters

**struct futex\_q \*q** the futex\_q  
**union futex\_key \*key** the key of the requeue target futex  
**struct futex\_hash\_bucket \*hb** the hash\_bucket of the requeue target futex

## Description

During futex\_requeue, with requeue\_pi=1, it is possible to acquire the target futex if it is uncontended or via a lock steal.

- 1) Set **q::key** to the requeue target futex key so the waiter can detect the wakeup on the right futex.
- 2) Dequeue **q** from the hash bucket.
- 3) Set **q::rt\_waiter** to NULL so the woken up task can detect atomic lock acquisition.

- 4) Set the q->lock\_ptr to the requeue target hb->lock for the case that the waiter has to fixup the pi state.
- 5) Complete the requeue state so the waiter can make progress. After this point the waiter task can return from the syscall immediately in case that the pi state does not have to be fixed up.
- 6) Wake the waiter task.

Must be called with both q->lock\_ptr and hb->lock held.

```
int futex_proxy_trylock_atomic(u32 __user *pifutex, struct futex_hash_bucket *hb1, struct futex_hash_bucket *hb2, union futex_key *key1, union futex_key *key2, struct futex_pi_state **ps, struct task_struct **exiting, int set_waiters)
```

Attempt an atomic lock for the top waiter

### Parameters

**u32 \_\_user \*pifutex** the user address of the to futex

**struct futex\_hash\_bucket \*hb1** the from futex hash bucket, must be locked by the caller

**struct futex\_hash\_bucket \*hb2** the to futex hash bucket, must be locked by the caller

**union futex\_key \*key1** the from futex key

**union futex\_key \*key2** the to futex key

**struct futex\_pi\_state \*\*ps** address to store the pi\_state pointer

**struct task\_struct \*\*exiting** Pointer to store the task pointer of the owner task which is in the middle of exiting

**int set\_waiters** force setting the FUTEX\_WAITERS bit (1) or not (0)

### Description

Try and get the lock on behalf of the top waiter if we can do it atomically. Wake the top waiter if we succeed. If the caller specified set\_waiters, then direct futex\_lock\_pi\_atomic() to force setting the FUTEX\_WAITERS bit. hb1 and hb2 must be held by the caller.

**exiting** is only set when the return value is -EBUSY. If so, this holds a refcount on the exiting task on return and the caller needs to drop it after waiting for the exit to complete.

### Return

- 0 - failed to acquire the lock atomically;
- >0 - acquired the lock, return value is vpid of the top\_waiter
- <0 - error

```
int futex_requeue(u32 __user *uaddr1, unsigned int flags, u32 __user *uaddr2, int nr_wake, int nr_requeue, u32 *cmpval, int requeue_pi)
```

Requeue waiters from uaddr1 to uaddr2

### Parameters

**u32 \_\_user \*uaddr1** source futex user address

**unsigned int flags** futex flags (FLAGS\_SHARED, etc.)

**u32 \_\_user \*uaddr2** target futex user address

**int nr\_wake** number of waiters to wake (must be 1 for requeue\_pi)  
**int nr\_requeue** number of waiters to requeue (0-INT\_MAX)  
**u32 \*cmpval uaddr1** expected value (or NULL)  
**int requeue\_pi** if we are attempting to requeue from a non-pi futex to a pi futex (pi to pi requeue is not supported)

## Description

Requeue waiters on uaddr1 to uaddr2. In the requeue\_pi case, try to acquire uaddr2 atomically on behalf of the top waiter.

## Return

- >=0 - on success, the number of tasks requeued or woken;
- <0 - on error

**int handle\_early\_requeue\_pi\_wakeup(struct futex\_hash\_bucket \*hb, struct futex\_q \*q, struct hrtimer\_sleeper \*timeout)**

Handle early wakeup on the initial futex

## Parameters

**struct futex\_hash\_bucket \*hb** the hash\_bucket futex\_q was originally enqueued on  
**struct futex\_q \*q** the futex\_q woken while waiting to be requeued  
**struct hrtimer\_sleeper \*timeout** the timeout associated with the wait (NULL if none)

## Description

Determine the cause for the early wakeup.

## Return

-EWOULDBLOCK or -ETIMEDOUT or -ERESTARTNOINTR

**int futex\_wait\_requeue\_pi(u32 \_\_user \*uaddr, unsigned int flags, u32 val, ktime\_t \*abs\_time, u32 bitset, u32 \_\_user \*uaddr2)**

Wait on uaddr and take uaddr2

## Parameters

**u32 \_\_user \*uaddr** the futex we initially wait on (non-pi)  
**unsigned int flags** futex flags (FLAGS\_SHARED, FLAGS\_CLOCKRT, etc.), they must be the same type, no requeueing from private to shared, etc.

**u32 val** the expected value of uaddr

**ktime\_t \*abs\_time** absolute timeout

**u32 bitset** 32 bit wakeup bitset set by userspace, defaults to all

**u32 \_\_user \*uaddr2** the pi futex we will take prior to returning to user-space

## Description

The caller will wait on uaddr and will be requeued by futex\_requeue() to uaddr2 which must be PI aware and unique from uaddr. Normal wakeup will wake on uaddr2 and complete the

acquisition of the rt\_mutex prior to returning to userspace. This ensures the rt\_mutex maintains an owner when it has waiters; without one, the pi logic would not know which task to boost/deboost, if there was a need to.

We call schedule in futex\_wait\_queue() when we enqueue and return there via the following-  
1) wakeup on uaddr2 after an atomic lock acquisition by futex\_requeue() 2) wakeup on uaddr2  
after a requeue 3) signal 4) timeout

If 3, cleanup and return -ERESTARTNOINTR.

If 2, we may then block on trying to take the rt\_mutex and return via: 5) successful lock 6)  
signal 7) timeout 8) other lock acquisition failure

If 6, return -EWOULDBLOCK (restarting the syscall would do the same).

If 4 or 7, we cleanup and return with -ETIMEDOUT.

### Return

- 0 - On success;
- <0 - On error

```
void futex_wait_queue(struct futex_hash_bucket *hb, struct futex_q *q, struct  
                          hrtimer_sleeper *timeout)
```

futex\_queue() and wait for wakeup, timeout, or signal

### Parameters

**struct futex\_hash\_bucket \*hb** the futex hash bucket, must be locked by the caller

**struct futex\_q \*q** the futex\_q to queue up on

**struct hrtimer\_sleeper \*timeout** the prepared hrtimer\_sleeper, or null for no timeout

```
int unqueue_multiple(struct futex_vector *v, int count)
```

Remove various futexes from their hash bucket

### Parameters

**struct futex\_vector \*v** The list of futexes to unqueue

**int count** Number of futexes in the list

### Description

Helper to unqueue a list of futexes. This can't fail.

### Return

- >=0 - Index of the last futex that was awoken;
- -1 - No futex was awoken

```
int futex_wait_multiple_setup(struct futex_vector *vs, int count, int *woken)
```

Prepare to wait and enqueue multiple futexes

### Parameters

**struct futex\_vector \*vs** The futex list to wait on

**int count** The size of the list

**int \*woken** Index of the last woken futex, if any. Used to notify the caller that it can return  
this index to userspace (return parameter)

## Description

Prepare multiple futexes in a single step and enqueue them. This may fail if the futex list is invalid or if any futex was already awoken. On success the task is ready to interruptible sleep.

## Return

- 1 - One of the futexes was woken by another thread
- 0 - Success
- <0 - -EFAULT, -EWOULDBLOCK or -EINVAL

```
void futex_sleep_multiple(struct futex_vector *vs, unsigned int count, struct hrtimer_sleeper *to)
```

Check sleeping conditions and sleep

## Parameters

**struct futex\_vector \*vs** List of futexes to wait for

**unsigned int count** Length of vs

**struct hrtimer\_sleeper \*to** Timeout

## Description

Sleep if and only if the timeout hasn't expired and no futex on the list has been woken up.

```
int futex_wait_multiple(struct futex_vector *vs, unsigned int count, struct hrtimer_sleeper *to)
```

Prepare to wait on and enqueue several futexes

## Parameters

**struct futex\_vector \*vs** The list of futexes to wait on

**unsigned int count** The number of objects

**struct hrtimer\_sleeper \*to** Timeout before giving up and returning to userspace

## Description

Entry point for the FUTEX\_WAIT\_MULTIPLE futex operation, this function sleeps on a group of futexes and returns on the first futex that is wake, or after the timeout has elapsed.

## Return

- >=0 - Hint to the futex that was awoken
- <0 - On error

```
int futex_wait_setup(u32 __user *uaddr, u32 val, unsigned int flags, struct futex_q *q, struct futex_hash_bucket **hb)
```

Prepare to wait on a futex

## Parameters

**u32 \_\_user \*uaddr** the futex userspace address

**u32 val** the expected value

**unsigned int flags** futex flags (FLAGS\_SHARED, etc.)

**struct futex\_q \*q** the associated futex\_q

**struct futex\_hash\_bucket \*\*hb** storage for hash\_bucket pointer to be returned to caller

### Description

Setup the futex\_q and locate the hash\_bucket. Get the futex value and compare it with the expected value. Handle atomic faults internally. Return with the hb lock held on success, and unlocked on failure.

### Return

- 0 - uaddr contains val and hb has been locked;
- <1 - -EFAULT or -EWOULDBLOCK (uaddr does not contain val) and hb is unlocked

### Approfondimenti

- Documentation/locking/spinlocks.rst: la guida di Linus Torvalds agli spinlock del kernel.
- Unix Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers.

L'introduzione alla sincronizzazione a livello di kernel di Curt Schimmel è davvero ottima (non è scritta per Linux, ma approssimativamente si adatta a tutte le situazioni). Il libro è costoso, ma vale ogni singolo spicciolo per capire la sincronizzazione nei sistemi multiprocessore. [ISBN: 0201633388]

### Ringraziamenti

Grazie a Telsa Gwynne per aver formattato questa guida in DocBook, averla pulita e aggiunto un po' di stile.

Grazie a Martin Pool, Philipp Rumpf, Stephen Rothwell, Paul Mackerras, Ruedi Aschwanden, Alan Cox, Manfred Spraul, Tim Waugh, Pete Zaitcev, James Morris, Robert Love, Paul McKenney, John Ashby per aver revisionato, corretto, maledetto e commentato.

Grazie alla congrega per non aver avuto alcuna influenza su questo documento.

### Glossario

**prelazione** Prima del kernel 2.5, o quando CONFIG\_PREEMPT non è impostato, i processi in contesto utente non si avvicendano nell'esecuzione (in pratica, il processo userà il processore fino al proprio termine, a meno che non ci siano delle interruzioni). Con l'aggiunta di CONFIG\_PREEMPT nella versione 2.5.4 questo è cambiato: quando si è in contesto utente, processi con una priorità maggiore possono subentrare nell'esecuzione: gli spinlock furono cambiati per disabilitare la prelazioni, anche su sistemi monoprocesso.

**bh** Bottom Half: per ragioni storiche, le funzioni che contengono '\_bh' nel loro nome ora si riferiscono a qualsiasi interruzione software; per esempio, spin\_lock\_bh() blocca qualsiasi interruzione software sul processore corrente. I *Bottom Halves* sono deprecati, e probabilmente verranno sostituiti dai tasklet. In un dato momento potrà esserci solo un *bottom half* in esecuzione.

**contesto d'interruzione** Non è il contesto utente: qui si processano le interruzioni hardware e software. La macro `in_interrupt()` ritorna vero.

**contesto utente** Il kernel che esegue qualcosa per conto di un particolare processo (per esempio una chiamata di sistema) o di un thread del kernel. Potete identificare il processo con la macro `current`. Da non confondere con lo spazio utente. Può essere interrotto sia da interruzioni software che hardware.

**interruzione hardware** Richiesta di interruzione hardware. `in_hardirq()` ritorna vero in un gestore d'interruzioni hardware.

**interruzione software / softirq** Gestore di interruzioni software: `in_hardirq()` ritorna falso; `in_softirq()` ritorna vero. I tasklet e le softirq sono entrambi considerati 'interruzioni software'.

In soldoni, un softirq è uno delle 32 interruzioni software che possono essere eseguite su più processori in contemporanea. A volte si usa per riferirsi anche ai tasklet (in pratica tutte le interruzioni software).

**monoprocessore / UP** (Uni-Processor) un solo processore, ovvero non è SMP. (`CONFIG_SMP=n`).

**multi-processore / SMP** (Symmetric Multi-Processor) kernel compilati per sistemi multi-processore (`CONFIG_SMP=y`).

**spazio utente** Un processo che esegue il proprio codice fuori dal kernel.

**tasklet** Un'interruzione software registrabile dinamicamente che ha la garanzia d'essere eseguita solo su un processore alla volta.

**timer** Un'interruzione software registrabile dinamicamente che viene eseguita (circa) in un determinato momento. Quando è in esecuzione è come un tasklet (infatti, sono chiamati da `TIMER_SOFTIRQ`).

## \* Documentazione della API del kernel

Questi manuali forniscono dettagli su come funzionano i sottosistemi del kernel dal punto di vista degli sviluppatori del kernel. Molte delle informazioni contenute in questi manuali sono prese direttamente dai file sorgenti, informazioni aggiuntive vengono aggiunte solo se necessarie (o almeno ci proviamo — probabilmente *non* tutto quello che è davvero necessario).

## Documentazione dell'API di base

### Utilità di base

**Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le [avvertenze](#).

**Original** ../../core-api/symbol-namespaces

**Translator** Federico Vaga <[federico.vaga@vaga.pv.it](mailto:federico.vaga@vaga.pv.it)>

## Spazio dei nomi dei simboli

Questo documento descrive come usare lo spazio dei nomi dei simboli per strutturare quello che viene esportato internamente al kernel grazie alle macro della famiglia EXPORT\_SYMBOL().

### 1. Introduzione

Lo spazio dei nomi dei simboli è stato introdotto come mezzo per strutturare l'API esposta internamente al kernel. Permette ai manutentori di un sottosistema di organizzare i simboli esportati in diversi spazi di nomi. Questo meccanismo è utile per la documentazione (pensate ad esempio allo spazio dei nomi SUBSYSTEM\_DEBUG) così come per limitare la disponibilità di un gruppo di simboli in altre parti del kernel. Ad oggi, i moduli che usano simboli esportati da uno spazio di nomi devono prima importare detto spazio. Altrimenti il kernel, a seconda della configurazione, potrebbe rifiutare di caricare il modulo o avvisare l'utente di un'importazione mancante.

### 2. Come definire uno spazio dei nomi dei simboli

I simboli possono essere esportati in spazi dei nomi usando diversi meccanismi. Tutti questi meccanismi cambiano il modo in cui EXPORT\_SYMBOL e simili vengono guidati verso la creazione di voci in ksymtab.

#### 2.1 Usare le macro EXPORT\_SYMBOL

In aggiunta alle macro EXPORT\_SYMBOL() e EXPORT\_SYMBOL\_GPL(), che permettono di esportare simboli del kernel nella rispettiva tabella, ci sono varianti che permettono di esportare simboli all'interno di uno spazio dei nomi: EXPORT\_SYMBOL\_NS() ed EXPORT\_SYMBOL\_NS\_GPL(). Queste macro richiedono un argomento aggiuntivo: lo spazio dei nomi. Tenete presente che per via dell'espansione delle macro questo argomento deve essere un simbolo di preprocessore. Per esempio per esportare il simbolo `usb_stor_suspend` nello spazio dei nomi `USB_STORAGE` usate:

```
EXPORT_SYMBOL_NS(usb_stor_suspend, USB_STORAGE);
```

Di conseguenza, nella tabella dei simboli del kernel ci sarà una voce rappresentata dalla struttura `kernel_symbol` che avrà il campo `namespace` (spazio dei nomi) impostato. Un simbolo esportato senza uno spazio dei nomi avrà questo campo impostato a `NULL`. Non esiste uno spazio dei nomi di base. Il programma `modpost` e il codice in `kernel/module/main.c` usano lo spazio dei nomi, rispettivamente, durante la compilazione e durante il caricamento di un modulo.

## 2.2 Usare il simbolo di preprocessore DEFAULT\_SYMBOL\_NAMESPACE

Definire lo spazio dei nomi per tutti i simboli di un sottosistema può essere logorante e di difficile manutenzione. Perciò è stato fornito un simbolo di preprocessore di base (DEFAULT\_SYMBOL\_NAMESPACE), che, se impostato, diventa lo spazio dei simboli di base per tutti gli usi di EXPORT\_SYMBOL() ed EXPORT\_SYMBOL\_GPL() che non specificano esplicitamente uno spazio dei nomi.

Ci sono molti modi per specificare questo simbolo di preprocessore e il loro uso dipende dalle preferenze del manutentore di un sottosistema. La prima possibilità è quella di definire il simbolo nel Makefile del sottosistema. Per esempio per esportare tutti i simboli definiti in usb-common nello spazio dei nomi USB\_COMMON, si può aggiungere la seguente linea in drivers/usb/common/Makefile:

```
ccflags-y += -DDEFAULT_SYMBOL_NAMESPACE=USB_COMMON
```

Questo cambierà tutte le macro EXPORT\_SYMBOL() ed EXPORT\_SYMBOL\_GPL(). Invece, un simbolo esportato con EXPORT\_SYMBOL\_NS() non verrà cambiato e il simbolo verrà esportato nello spazio dei nomi indicato.

Una seconda possibilità è quella di definire il simbolo di preprocessore direttamente nei file da compilare. L'esempio precedente diventerebbe:

```
#undef DEFAULT_SYMBOL_NAMESPACE
#define DEFAULT_SYMBOL_NAMESPACE USB_COMMON
```

Questo va messo prima di un qualsiasi uso di EXPORT\_SYMBOL.

## 3. Come usare i simboli esportati attraverso uno spazio dei nomi

Per usare i simboli esportati da uno spazio dei nomi, i moduli del kernel devono esplicitamente importare il relativo spazio dei nomi; altrimenti il kernel potrebbe rifiutarsi di caricare il modulo. Il codice del modulo deve usare la macro MODULE\_IMPORT\_NS per importare lo spazio dei nomi che contiene i simboli desiderati. Per esempio un modulo che usa il simbolo usb\_stor\_suspend deve importare lo spazio dei nomi USB\_STORAGE usando la seguente dichiarazione:

```
MODULE_IMPORT_NS(USB_STORAGE);
```

Questo creerà un'etichetta modinfo per ogni spazio dei nomi importato. Un risvolto di questo fatto è che gli spazi dei nomi importati da un modulo possono essere ispezionati tramite modinfo:

```
$ modinfo drivers/usb/storage/ums-karma.ko
[...]
import_ns:      USB_STORAGE
[...]
```

Si consiglia di posizionare la dichiarazione MODULE\_IMPORT\_NS() vicino ai metadati del modulo come MODULE\_AUTHOR() o MODULE\_LICENSE(). Fate riferimento alla sezione 5. per creare automaticamente le importazioni mancanti.

### 4. Caricare moduli che usano simboli provenienti da spazi dei nomi

Quando un modulo viene caricato (per esempio usando `insmod`), il kernel verificherà la disponibilità di ogni simbolo usato e se lo spazio dei nomi che potrebbe contenerli è stato importato. Il comportamento di base del kernel è di rifiutarsi di caricare quei moduli che non importano tutti gli spazi dei nomi necessari. L'errore verrà annotato e il caricamento fallirà con l'errore `EINVAL`. Per caricare i moduli che non soddisfano questo requisito esiste un'opzione di configurazione: impostare `MODULE_ALLOW_MISSING_NAMESPACE_IMPORTS=y` caricherà i moduli comunque ma emetterà un avviso.

### 5. Creare automaticamente la dichiarazione MODULE\_IMPORT\_NS

La mancanza di un'importazione può essere individuata facilmente al momento della compilazione. Infatti, modpost emetterà un avviso se il modulo usa un simbolo da uno spazio dei nomi che non è stato importato. La dichiarazione `MODULE_IMPORT_NS()` viene solitamente aggiunta in un posto ben definito (assieme agli altri metadati del modulo). Per facilitare la vita di chi scrive moduli (e i manutentori di sottosistemi), esistono uno script e un target make per correggere le importazioni mancanti. Questo può essere fatto con:

```
$ make nsdeps
```

Lo scenario tipico di chi scrive un modulo potrebbe essere:

- scrivere codice che dipende da un simbolo appartenente ad uno spazio dei nomi non importato
- eseguire ``make``
- aver notato un avviso da modpost che parla di un'importazione mancante
- eseguire ``make nsdeps`` per aggiungere import nel posto giusto

Per i manutentori di sottosistemi che vogliono aggiungere uno spazio dei nomi, l'approccio è simile. Di nuovo, eseguendo `make nsdeps` aggiungerà le importazioni mancanti nei moduli inclusi nel kernel:

- spostare o aggiungere simboli ad uno spazio dei nomi (per esempio usando `EXPORT_SYMBOL_NS()`)
- eseguire ``make`` (preferibilmente con `allmodconfig` per coprire tutti i moduli del kernel)
- aver notato un avviso da modpost che parla di un'importazione mancante
- eseguire ``make nsdeps`` per aggiungere import nel posto giusto

Potete anche eseguire `nsdeps` per moduli esterni. Solitamente si usa così:

```
$ make -C <path_to_kernel_src> M=$PWD nsdeps
```

### \* Documentazione specifica per architettura

Questi manuali forniscono dettagli di programmazione per le diverse implementazioni d'architettura.

**Warning:** TODO ancora da tradurre



## **한국어 번역**

NOTE: This is a version of Documentation/process/howto.rst translated into korean. This document is maintained by Minchan Kim <[minchan@kernel.org](mailto:minchan@kernel.org)>. If you find any difference between this document and the original file or a problem with the translation, please contact the maintainer of this file.

Please also note that the purpose of this file is to be easier to read for non English (read: korean) speakers and is not intended as a fork. So if you have any comments or updates for this file please try to update the original English file first.

---

이 문서는 Documentation/process/howto.rst 의 한글 번역입니다.

역자 : 김민찬 <[minchan@kernel.org](mailto:minchan@kernel.org)> 감수 : 이제이미 <[jamee.lee@samsung.com](mailto:jamee.lee@samsung.com)>

---

### **\* 어떻게 리눅스 커널 개발을 하는가**

이 문서는 커널 개발에 있어 가장 중요한 문서이다. 이 문서는 리눅스 커널 개발자가 되는 법과 리눅스 커널 개발 커뮤니티와 일하는 법을 담고있다. 커널 프로그래밍의 기술적인 측면과 관련된 내용들은 포함하지 않으려고 하였지만 올바른 길로 여러분을 안내하는 데는 도움이 될 것이다.

이 문서에서 오래된 것을 발견하면 문서의 아래쪽에 나열된 메인테이너에게 패치를 보내달라.

### **\* 소개**

자, 여러분은 리눅스 커널 개발자가 되는 법을 배우고 싶은가? 아니면 상사로부터 “이 장치를 위한 리눅스 드라이버를 작성하시오”라는 말을 들었는가? 이 문서의 목적은 여러분이 겪게 될 과정과 커뮤니티와 협력하는 법을 조언하여 여러분의 목적을 달성하기 위해 필요한 것 모두를 알려주기 위함이다.

커널은 대부분은 C로 작성되어 있고 몇몇 아키텍쳐의 의존적인 부분은 어셈블리로 작성되어 있다. 커널 개발을 위해 C를 잘 이해하고 있어야 한다. 여러분이 특정 아키텍처의 low-level 개발을 할 것이 아니라면 어셈블리(특정 아키텍쳐)는 잘 알아야 할 필요는 없다. 다음의 참고서적들은 기본에 충실한 C 교육이나 수년간의 경험에 견주지는 못하지만 적어도 참고 용도로는 좋을 것이다

- “The C Programming Language” by Kernighan and Ritchie [Prentice Hall]
- “Practical C Programming” by Steve Oualline [O’Reilly]
- “C: A Reference Manual” by Harbison and Steele [Prentice Hall]

커널은 GNU C 와 GNU 툴체인을 사용하여 작성되었다. 이 툴들은 ISO C89 표준을 따르는 반면 표준에 있지 않은 많은 확장기능도 가지고 있다. 커널은 표준 C 라이브러리와는 관계없이 freestanding C 환경이어서 C 표준의 일부는 지원되지 않는다. 임의의 long long 나누기나 floating point 는 지원되지 않는다. 때론 이런 이유로 커널이 그런 확장 기능을 가진 툴체인을 가지고 만들어졌다는 것이 이해하기 어려울 수도 있고 게다가 불행하게도 그런 것을 정확하게 설명하는 어떤 참고문서도 있지 않다. 정보를 얻기 위해서는 *gcc info (info gcc)* 페이지를 살펴보라.

여러분은 기존의 개발 커뮤니티와 협력하는 법을 배우려고 하고 있다는 것을 기억하라. 코딩, 스타일, 함수에 관한 훌륭한 표준을 가진 사람들이 모인 다양한 그룹이 있다. 이 표준들은 오랜동안 크고 지역적으로 분산된 팀들에 의해 가장 좋은 방법으로 일하기 위하여 찾은 것을 기초로 만들어져 왔다. 그 표준들은 문서화가 잘 되어있기 때문에 가능한한 미리 많은 표준들에 관하여 배우려고 시도하라. 다른 사람들은 여러분이나 여러분의 회사가 일하는 방식에 적응하는 것을 원하지는 않는다.

### \* 법적 문제

리눅스 커널 소스 코드는 GPL 로 배포 (release) 되었다. 소스트리의 메인 디렉토리에 있는 라이센스에 관하여 상세하게 쓰여 있는 COPYING 이라는 파일을 봐라. 리눅스 커널 라이센싱 규칙과 소스 코드 안의 SPDX 식별자 사용법은 Documentation/process/license-rules.rst 에 설명되어 있다. 여러분이 라이센스에 관한 더 깊은 문제를 가지고 있다면 리눅스 커널 메일링 리스트에 묻지말고 변호사와 연락하라. 메일링 리스트들에 있는 사람들은 변호사가 아니기 때문에 법적 문제에 관하여 그들의 말에 의지해서는 안된다.

GPL 에 관한 갖은 질문들과 답변들은 다음을 참조하라.

<https://www.gnu.org/licenses/gpl-faq.html>

### \* 문서

리눅스 커널 소스 트리는 커널 커뮤니티와 협력하는 법을 배우기위해 훌륭한 다양한 문서들을 가지고 있다. 새로운 기능들이 커널에 들어가게 될 때, 그 기능을 어떻게 사용하는지에 관한 설명을 위하여 새로운 문서 파일을 추가하는 것을 권장한다. 커널이 유저스페이스로 노출하는 인터페이스를 변경하게 되면 변경을 설명하는 메뉴얼 페이지들에 대한 패치나 정보를 [mtk.manpages@gmail.com](mailto:mtk.manpages@gmail.com) 의 메인테이너에게 보낼 것을 권장한다.

다음은 커널 소스 트리에 있는 읽어야 할 파일들의 리스트이다.

**Documentation/admin-guide/README.rst** 이 파일은 리눅스 커널에 관하여 간단한 배경 설명과 커널을 설정하고 빌드하기 위해 필요한 것을 설명한다. 커널에 입문하는 사람들은 여기서 시작해야 한다.

**Documentation/process/changes.rst** 이 파일은 커널을 성공적으로 빌드하고 실행시키기 위해 필요한 다양한 소프트웨어 패키지들의 최소 버전을 나열한다.

**Documentation/process/coding-style.rst** 이 문서는 리눅스 커널 코딩 스타일과 그렇게 한 몇몇 이유를 설명한다. 모든 새로운 코드는 이 문서에 가이드라인들을 따라야 한다. 대부분의 메인테이너들은 이 규칙을 따르는 패치들만을 받아들일 것이고 많은 사람들이 그 패치가 올바른 스타일일 경우만 코드를 검토할 것이다.

## Documentation/process/submitting-patches.rst 와 Documentation/process/submitting-d

이 파일들은 성공적으로 패치를 만들고 보내는 법을 다음의 내용들로 굉장히 상세히 설명하고 있다 (그러나 다음으로 한정되진 않는다).

- Email 내용들
- Email 양식
- 그것을 누구에게 보낼지

이러한 규칙들을 따르는 것이 성공 (역자주: 패치가 받아들여지는 것) 을 보장하진 않는다 (왜냐하면 모든 패치들은 내용과 스타일에 관하여 면밀히 검토되기 때문이다). 그러나 규칙을 따르지 않는다면 거의 성공하지도 못할 것이다.

올바른 패치들을 만드는 법에 관한 훌륭한 다른 문서들이 있다.

“The Perfect Patch” <https://www.ozlabs.org/~akpm/stuff/tpp.txt>

“Linux kernel patch submission format” <https://web.archive.org/web/20180829112450/http://linux.yyz.us/patch-format.html>

**Documentation/process/stable-api-nonsense.rst** 이 문서는 의도적으로 커널이 불변하는 API 를 갖지 않도록 결정한 이유를 설명하며 다음과 같은 것들을 포함한다.

- 서브시스템 shim-layer(호환성을 위해?)
- 운영체제들간의 드라이버 이식성
- 커널 소스 트리내에 빠른 변화를 늦추는 것 (또는 빠른 변화를 막는 것)

이 문서는 리눅스 개발 철학을 이해하는데 필수적이며 다른 운영체제에서 리눅스로 전향하는 사람들에게는 매우 중요하다.

**Documentation/admin-guide/security-bugs.rst** 여러분들이 리눅스 커널의 보안 문제를 발견했다고 생각한다면 이 문서에 나온 단계에 따라서 커널 개발자들에게 알리고 그 문제를 해결할 수 있도록 도와 달라.

**Documentation/process/management-style.rst** 이 문서는 리눅스 커널 메인테이너들이 그들의 방법론에 녹아 있는 정신을 어떻게 공유하고 운영하는지를 설명한다. 이것은 커널 개발에 입문하는 모든 사람들 (또는 커널 개발에 작은 호기심이라도 있는 사람들) 이 읽어야 할 중요한 문서이다. 왜냐하면 이 문서는 커널 메인테이너들의 독특한 행동에 관하여 흔히 있는 오해들과 혼란들을 해소하고 있기 때문이다.

**Documentation/process/stable-kernel-rules.rst** 이 문서는 안정적인 커널 배포가 이루어지는 규칙을 설명하고 있으며 여러분들이 이러한 배포들 중 하나에 변경을 하길 원한다면 무

엇을 해야 하는지를 설명한다.

**Documentation/process/kernel-docs.rst** 커널 개발에 관계된 외부 문서의 리스트이다.

커널 내의 포함된 문서들 중에 여러분이 찾고 싶은 문서를 발견하지 못할 경우 이 리스트를 살펴보라.

**Documentation/process/applying-patches.rst** 패치가 무엇이며 그것을 커널의 다른 개발 브랜치들에 어떻게 적용하는지에 관하여 자세히 설명하고 있는 좋은 입문서이다.

커널은 소스 코드 그 자체에서 또는 이것과 같은 ReStructuredText 마크업 (ReST) 을 통해 자동적으로 만들 어질 수 있는 많은 문서들을 가지고 있다. 이것은 커널 내의 API 에 대한 모든 설명, 그리고 락킹을 올바르게 처리하는 법에 관한 규칙을 포함하고 있다.

모든 그런 문서들은 커널 소스 디렉토리에서 다음 커맨드를 실행하는 것을 통해 PDF 나 HTML 의 형태로 만들 어질 수 있다:

```
make pdfdocs  
make htmldocs
```

ReST 마크업을 사용하는 문서들은 Documentation/output 에 생성된다. 해당 문서들은 다음의 커맨드를 사용하면 LaTeX 이나 ePub 로도 만들어질 수 있다:

```
make latexdocs  
make epubdocs
```

### \* 커널 개발자가 되는 것

여러분이 리눅스 커널 개발에 관하여 아무것도 모른다면 Linux KernelNewbies 프로젝트를 봐야 한다.

<https://kernelnewbies.org>

그곳은 거의 모든 종류의 기본적인 커널 개발 질문들 (질문하기 전에 먼저 아카이브를 찾아봐라. 과거에 이미 답변되었을 수도 있다) 을 할 수 있는 도움이 될만한 메일링 리스트가 있다. 또한 실시간으로 질문 할 수 있는 IRC 채널도 가지고 있으며 리눅스 커널 개발을 배우는데 유용한 문서들을 보유하고 있다.

웹사이트는 코드구성, 서브시스템들, 그리고 현재 프로젝트들 (트리 내, 외부에 존재하는) 에 관한 기본적인 정보들을 가지고 있다. 또한 그곳은 커널 컴파일이나 패치를 하는 법과 같은 기본적인 것들을 설명한다.

여러분이 어디서 시작해야 할진 모르지만 커널 개발 커뮤니티에 참여할 수 있는 일들을 찾길 원한다면 리눅스 커널 Janitor 프로젝트를 살펴봐라.

<https://kernelnewbies.org/KernelJanitors>

그곳은 시작하기에 훌륭한 장소이다. 그곳은 리눅스 커널 소스 트리내에 간단히 정리되고 수정될 수 있는 문제들에 관하여 설명한다. 여러분은 이 프로젝트를 대표하는 개발자들과 일하면서 자신의 패치를 리눅스 커널 트리에 반영하기 위한 기본적인 것들을 배우게 될것이며 여러분이 아직 아이디어를 가지고 있지 않다면 다음에 무엇을 해야할지에 관한 방향을 배울 수 있을 것이다.

리눅스 커널 코드에 실제 변경을 하기 전에 반드시 그 코드가 어떻게 동작하는지 이해하고 있어야 한다. 코드를 분석하기 위하여 특정한 툴의 도움을 빌려서라도 코드를 직접 읽는 것보다 좋은 것은 없다 (대부분의 자잘한 부

분들은 잘 코멘트되어 있다). 그런 툴들 중에 특히 추천할만한 것은 Linux Cross-Reference project이며 그것은 자기 참조 방식이며 소스코드를 인덱스된 웹 페이지들의 형태로 보여준다. 최신의 멋진 커널 코드 저장소는 다음을 통하여 참조할 수 있다.

<https://elixir.bootlin.com/>

## \* 개발 프로세스

리눅스 커널 개발 프로세스는 현재 몇몇 다른 메인 커널 “브랜치들”과 서브시스템에 특화된 커널 브랜치들로 구성된다. 몇몇 다른 메인 브랜치들은 다음과 같다.

- 리눅스의 메인라인 트리
- 여러 메이저 넘버를 갖는 다양한 안정된 커널 트리들
- 서브시스템을 위한 커널 트리들
- 통합 테스트를 위한 linux-next 커널 트리

## 메인라인 트리

메인라인 트리는 Linus Torvalds 가 관리하며 <https://kernel.org> 또는 소스 저장소에서 참조될 수 있다. 개발 프로세스는 다음과 같다.

- 새로운 커널이 배포되자마자 2 주의 시간이 주어진다. 이 기간동은 메인테이너들은 큰 diff 들을 Linus에게 제출할 수 있다. 대개 이 패치들은 몇 주 동안 linux-next 커널내에 이미 있었던 것들이다. 큰 변경들을 제출하는 데 선호되는 방법은 git(커널의 소스 관리 툴, 더 많은 정보들은 <https://git-scm.com/> 에서 참조할 수 있다)를 사용하는 것이지만 순수한 패치파일의 형식으로 보내는 것도 무관하다.
- 2 주 후에 -rc1 커널이 릴리즈되며 여기서부터의 주안점은 새로운 커널을 가능한한 안정되게 하는 것이다. 이 시점에서의 대부분의 패치들은 회귀 (역자주: 이전에는 존재하지 않았지만 새로운 기능추가나 변경으로 인해 생겨난 버그) 를 고쳐야 한다. 이전부터 존재한 버그는 회귀가 아니므로, 그런 버그에 대한 수정사항은 중요한 경우에만 보내져야 한다. 완전히 새로운 드라이버 (혹은 파일시스템) 는 -rc1 이후에만 받아들여진다는 것을 기억해라. 왜냐하면 변경이 자체내에서만 발생하고 추가된 코드가 드라이버 외부의 다른 부분에는 영향을 주지 않으므로 그런 변경은 회귀를 일으킬 만한 위험을 가지고 있지 않기 때문이다. -rc1 이 배포된 이후에 git 를 사용하여 패치들을 Linus에게 보낼수 있지만 패치들은 공식적인 메일링 리스트로 보내서 검토를 받을 필요가 있다.
- 새로운 -rc 는 Linus 가 현재 git tree 가 테스트 하기에 충분히 안정된 상태에 있다고 판단될 때마다 배포된다. 목표는 새로운 -rc 커널을 매주 배포하는 것이다.
- 이러한 프로세스는 커널이 “준비 (ready)” 되었다고 여겨질때까지 계속된다. 프로세스는 대체로 6 주간 지속된다.

커널 배포에 있어서 언급할만한 가치가 있는 리눅스 커널 메일링 리스트의 Andrew Morton 의 글이 있다.

“커널이 언제 배포될지는 아무도 모른다. 왜냐하면 배포는 알려진 버그의 상황에 따라 배포되는 것 이지 미리정해 놓은 시간에 따라 배포되는 것은 아니기 때문이다.”

### 여러 메이저 넘버를 갖는 다양한 안정된 커널 트리들

세개의 버전 넘버로 이루어진 버전의 커널들은 -stable 커널들이다. 그것들은 해당 메이저 메인라인 릴리즈에서 발견된 큰 허귀들이나 보안 문제들 중 비교적 작고 중요한 수정들을 포함한다. 주요 stable 시리즈 릴리즈는 세번째 버전 넘버를 증가시키며 앞의 두 버전 넘버는 그대로 유지한다.

이것은 가장 최근의 안정적인 커널을 원하는 사용자에게 추천되는 브랜치이며, 개발/실험적 버전을 테스트하는 것을 돋고자 하는 사용자들과는 별로 관련이 없다.

-stable 트리들은 “stable” 팀 <[stable@vger.kernel.org](mailto:stable@vger.kernel.org)> 에 의해 관리되며 거의 매번 격주로 배포된다.

커널 트리 문서들 내의 Documentation/process/stable-kernel-rules.rst 파일은 어떤 종류의 변경들이 -stable 트리로 들어왔는지와 배포 프로세스가 어떻게 진행되는지를 설명한다.

### 서브시스템 커널 트리들

다양한 커널 서브시스템의 메인테이너들—그리고 많은 커널 서브시스템 개발자들—은 그들의 현재 개발 상태를 소스 저장소로 노출한다. 이를 통해 다른 사람들도 커널의 다른 영역에 어떤 변화가 이루어지고 있는지 알 수 있다. 급속히 개발이 진행되는 영역이 있고 그렇지 않은 영역이 있으므로, 개발자는 다른 개발자가 제출한 수정 사항과 자신의 수정사항의 충돌이나 동일한 일을 동시에 두사람이 따로 진행하는 사태를 방지하기 위해 급속히 개발이 진행되고 있는 영역에 작업의 베이스를 맞춰줄 것이 요구된다.

대부분의 이러한 저장소는 git 트리지만, git 이 아닌 SCM 으로 관리되거나, quilt 시리즈로 제공되는 패치들도 존재한다. 이러한 서브시스템 저장소들은 MAINTAINERS 파일에 나열되어 있다. 대부분은 <https://git.kernel.org> 에서 볼 수 있다.

제안된 패치는 서브시스템 트리에 커밋되기 전에 메일링 리스트를 통해 리뷰된다 (아래의 관련 섹션을 참고하기 바란다). 일부 커널 서브시스템의 경우, 이 리뷰 프로세스는 patchwork 라는 도구를 통해 추적된다. patchwork 은 등록된 패치와 패치에 대한 코멘트, 패치의 버전을 볼 수 있는 웹 인터페이스를 제공하고, 메인 테이너는 패치를 리뷰 중, 리뷰 통과, 또는 반려됨으로 표시할 수 있다. 대부분의 이러한 patchwork 사이트는 <https://patchwork.kernel.org/> 에 나열되어 있다.

### 통합 테스트를 위한 linux-next 커널 트리

서브시스템 트리들의 변경사항들은 mainline 트리로 들어오기 전에 통합 테스트를 거쳐야 한다. 이런 목적으로, 모든 서브시스템 트리의 변경사항을 거의 매일 받아가는 특수한 테스트 저장소가 존재한다:

<https://git.kernel.org/?p=linux/kernel/git/next/linux-next.git>

이런 식으로, linux-next 커널을 통해 다음 머지 기간에 메인라인 커널에 어떤 변경이 가해질 것인지 간략히 알 수 있다. 모험심 강한 테스터라면 linux-next 커널에서 테스트를 수행하는 것도 좋을 것이다.

## \* 버그 보고

메인 커널 소스 디렉토리에 있는 ‘Documentation/admin-guide/reporting-issues.rst’ 파일은 커널 버그라고 생각되는 것을 어떻게 보고하면 되는지, 그리고 문제를 추적하기 위해서 커널 개발자들이 필요로 하는 정보가 무엇들인지를 상세히 설명하고 있다.

## \* 버그 리포트들의 관리

여러분의 해킹 기술을 연습하는 가장 좋은 방법 중의 하는 다른 사람들이 보고한 버그들을 수정하는 것이다. 여러분은 커널을 더욱 안정화시키는데 도움을 줄 뿐만이 아니라 실제있는 문제들을 수정하는 법을 배우게 되고 그와 함께 여러분들의 기술은 향상될 것이며 다른 개발자들이 여러분의 존재에 대해 알게 될 것이다. 버그를 수정하는 것은 개발자들 사이에서 점수를 얻을 수 있는 가장 좋은 방법중의 하나이다. 왜냐하면 많은 사람들은 다른 사람들의 버그들을 수정하기 위하여 시간을 낭비하지 않기 때문이다.

이미 보고된 버그 리포트들을 가지고 작업하기 위해서는 여러분이 관심있는 서브시스템을 찾아라. 해당 서브시스템의 버그들이 어디로 리포트 되는지 MAINTAINERS 파일을 체크하라; 그건 대부분 메일링 리스트이고, 가끔은 버그 추적 시스템이다. 그 장소에 있는 최근 버그 리포트 기록들을 검색하고 여러분이 보기에 적합하다 싶은 것을 도와라. 여러분은 버그 리포트를 위해 <https://bugzilla.kernel.org> 를 체크하고자 할 수도 있다; 소수의 커널 서브시스템들만이 버그 신고와 추적을 위해 해당 시스템을 실제로 사용하고 있지만, 전체 커널의 버그들이 그곳에 정리된다.

## \* 메일링 리스트들

위의 몇몇 문서들이 설명하였지만 핵심 커널 개발자들의 대다수는 리눅스 커널 메일링 리스트에 참여하고 있다. 리스트에 등록하고 해지하는 방법에 관한 자세한 사항은 다음에서 참조할 수 있다.

<http://vger.kernel.org/vger-lists.html#linux-kernel>

웹상의 많은 다른 곳에도 메일링 리스트의 아카이브들이 있다. 이러한 아카이브들을 찾으려면 검색 엔진을 사용하라. 예를 들어:

<http://dir.gmane.org/gmane.linux.kernel>

여러분이 새로운 문제에 관해 리스트에 올리기 전에 말하고 싶은 주제에 관한 것을 아카이브에서 먼저 찾아보기를 강력히 권장한다. 이미 상세하게 토론된 많은 것들이 메일링 리스트의 아카이브에 기록되어 있다.

각각의 커널 서브시스템들의 대부분은 자신들의 개발에 관한 노력들로 이루어진 분리된 메일링 리스트를 따로 가지고 있다. 다른 그룹들이 무슨 리스트를 가지고 있는지는 MAINTAINERS 파일을 참조하라.

많은 리스트들은 kernel.org 에서 호스트되고 있다. 그 정보들은 다음에서 참조될 수 있다.

<http://vger.kernel.org/vger-lists.html>

리스트들을 사용할 때는 올바른 예절을 따를 것을 유념해라. 대단하진 않지만 다음 URL 은 리스트 (혹은 모든 리스트) 와 대화하는 몇몇 간단한 가이드라인을 가지고 있다.

<http://www.albion.com/netiquette/>

여러 사람들이 여러분의 메일에 응답한다면 CC: 즉 수신 리스트는 꽤 커지게 될 것이다. 아무 이유 없이 CC에서 어떤 사람도 제거하거나 리스트 주소로만 회신하지 마라. 메일을 보낸 사람으로서 하나를 받고 리스트로부터 또 하나를 받아 두번 받는 것에 익숙하여 있으니 mail-header를 조작하려고 하지 말아라. 사람들은 그런 것을 좋아하지 않을 것이다.

여러분의 회신의 문맥을 원래대로 유지해야 한다. 여러분들의 회신의 윗부분에 “John 커널해커는 작성했다….”를 유지하며 여러분들의 의견을 그 메일의 윗부분에 작성하지 말고 각 인용한 단락들 사이에 넣어라.

여러분들이 패치들을 메일에 넣는다면 그것들은 Documentation/process/submitting-patches.rst에 나와 있는데 명백히 (plain) 읽을 수 있는 텍스트여야 한다. 커널 개발자들은 첨부파일이나 압축된 패치들을 원하지 않는다. 그들은 여러분들의 패치의 각 라인 단위로 코멘트를 하길 원하며 압축하거나 첨부하지 않고 보내는 것이 그렇게 할 수 있는 유일한 방법이다. 여러분들이 사용하는 메일 프로그램이 스페이스나 탭 문자들을 조작하지 않는지 확인하라. 가장 좋은 첫 테스트는 메일을 자신에게 보내보고 스스로 그 패치를 적용해보라. 그것이 동작하지 않는다면 여러분의 메일 프로그램을 고치던가 제대로 동작하는 프로그램으로 바꾸어라.

무엇보다도 메일링 리스트의 다른 구독자들에게 보여주려 한다는 것을 기억하라.

### \* 커뮤니티와 협력하는 법

커널 커뮤니티의 목적은 가능한한 가장 좋은 커널을 제공하는 것이다. 여러분이 받아들여질 패치를 제출하게 되면 그 패치의 기술적인 이점으로 검토될 것이다. 그럼 여러분들은 무엇을 기대하고 있어야 하는가?

- 비판
- 의견
- 변경을 위한 요구
- 당위성을 위한 요구
- 침목

기억하라. 이것들은 여러분의 패치가 커널로 들어가기 위한 과정이다. 여러분의 패치들은 비판과 다른 의견을 받을 수 있고 그것을 기술적인 레벨로 평가하고 재작업하거나 또는 왜 수정하면 안되는지에 관하여 명료하고 간결한 이유를 말할 수 있어야 한다. 여러분이 제출한 것에 어떤 응답도 있지 않다면 몇 일을 기다려보고 다시 시도해라. 때론 너무 많은 메일들 속에 묻혀버리기도 한다.

여러분은 무엇을 해서는 안되는가?

- 여러분의 패치가 아무 질문 없이 받아들여지기를 기대하는 것
- 방어적이 되는 것
- 의견을 무시하는 것
- 요청된 변경을 하지 않고 패치를 다시 제출하는 것

가능한한 가장 좋은 기술적인 해답을 찾고 있는 커뮤니티에서는 항상 어떤 패치가 얼마나 좋은지에 관하여 다른 의견들이 있을 수 있다. 여러분은 협조적이어야 하고 기꺼이 여러분의 생각을 커널 내에 맞추어야 한다. 아니면 적어도 여러분의 것이 가치있다는 것을 증명하여야 한다. 잘못된 것도 여러분이 올바른 방향의 해결책으로 이끌어갈 의지가 있다면 받아들여질 것이라는 점을 기억하라.

여러분의 첫 패치에 여러분이 수정해야하는 십여개 정도의 회신이 오는 경우도 흔하다. 이것은 여러분의 패치가 받아들여지지 않을 것이라는 것을 의미하는 것이 아니고 개인적으로 여러분에게 감정이 있어서 그려는 것도 아니다. 간단히 여러분의 패치에 제기된 문제들을 수정하고 그것을 다시 보내라.

## \* 커널 커뮤니티와 기업 조직간의 차이점

커널 커뮤니티는 가장 전통적인 회사의 개발 환경과는 다르다. 여기에 여러분들의 문제를 피하기 위한 목록이 있다.

여러분들이 제안한 변경들에 관하여 말할 때 좋은 것들:

- “이것은 여러 문제들을 해결합니다.”
- “이것은 2000 라인의 코드를 줄입니다.”
- “이것은 내가 말하려는 것에 관해 설명하는 패치입니다.”
- “나는 5 개의 다른 아키텍쳐에서 그것을 테스트 했으므로…”
- “여기에 일련의 작은 패치들이 있으므로…”
- “이것은 일반적인 머신에서 성능을 향상함으로…”

여러분들이 말할 때 피해야 할 좋지 않은 것들:

- “우리는 그것을 AIX/ptx/Solaris에서 이러한 방법으로 했다. 그러므로 그것은 좋은 것임에 틀림없다…”
- “나는 20년 동안 이것을 해왔다. 그러므로…”
- “이것은 돈을 벌기 위해 나의 회사가 필요로 하는 것이다.”
- “이것은 우리의 엔터프라이즈 상품 라인을 위한 것이다.”
- “여기에 나의 생각을 말하고 있는 1000 페이지 설계 문서가 있다.”
- “나는 6달 동안 이것을 했으니…”
- “여기에 5000 라인 짜리 패치가 있으니…”
- “나는 현재 뒤죽박죽인 것을 재작성했다. 그리고 여기에…”
- “나는 마감시한을 가지고 있으므로 이 패치는 지금 적용될 필요가 있다.”

커널 커뮤니티가 전통적인 소프트웨어 엔지니어링 개발 환경들과 또 다른 점은 얼굴을 보지 않고 일한다는 점이다. 이메일과 irc를 대화의 주요수단으로 사용하는 것의 한가지 장점은 성별이나 인종의 차별이 없다는 것이다. 리눅스 커널의 작업 환경에서는 단지 이메일 주소만 알 수 있기 때문에 여성과 소수 민족들도 모두 받아들여진다. 국제적으로 일하게 되는 측면은 사람의 이름에 근거하여 성별을 추측할 수 없게 하기 때문에 차별을 없애는 데 도움을 준다. Andrea라는 이름을 가진 남자와 Pat이라는 이름을 가진 여자가 있을 수도 있는 것이다. 리눅스 커널에서 작업하며 생각을 표현해왔던 대부분의 여성들은 긍정적인 경험을 가지고 있다.

언어 장벽은 영어에 익숙하지 않은 몇몇 사람들에게 문제가 될 수도 있다. 언어의 훌륭한 구사는 메일링 리스트에서 올바르게 자신의 생각을 표현하기 위하여 필요하다. 그래서 여러분은 이메일을 보내기 전에 영어를 올바르게 사용하고 있는지를 체크하는 것이 바람직하다.

### \* 여러분의 변경을 나누어라

리눅스 커널 커뮤니티는 한꺼번에 굉장히 큰 코드의 묶음 (chunk) 을 쉽게 받아들이지 않는다. 변경은 적절하게 소개되고, 검토되고, 각각의 부분으로 작게 나누어져야 한다. 이것은 회사에서 하는 것과는 정확히 반대되는 것이다. 여러분들의 제안은 개발 초기에 일찍이 소개되어야 한다. 그래서 여러분들은 자신이 하고 있는 것에 관하여 피드백을 받을 수 있게 된다. 커뮤니티가 여러분들이 커뮤니티와 함께 일하고 있다는 것을 느끼도록 만들고 커뮤니티가 여러분의 기능을 위한 쓰레기 장으로써 사용되지 않고 있다는 것을 느끼게 하자. 그러나 메일링 리스트에 한번에 50 개의 이메일을 보내지는 말아라. 여러분들의 일련의 패치들은 항상 더 작아야 한다.

패치를 나누는 이유는 다음과 같다.

- 1) 작은 패치들은 여러분의 패치들이 적용될 수 있는 확률을 높여준다. 왜냐하면 다른 사람들은 정확성을 검증하기 위하여 많은 시간과 노력을 들이기를 원하지 않는다. 5 줄의 패치는 메인테이너가 거의 몇 초간 힐끗 보면 적용될 수 있다. 그러나 500 줄의 패치는 정확성을 검토하기 위하여 몇시간이 걸릴 수도 있다 (걸리는 시간은 패치의 크기 혹은 다른 것에 비례하여 기하급수적으로 늘어난다).  
패치를 작게 만드는 것은 무엇인가 잘못되었을 때 디버그하는 것을 쉽게 만든다. 즉, 그렇게 만드는 것은 매우 큰 패치를 적용한 후에 조사하는 것 보다 작은 패치를 적용한 후에 (그리고 몇몇의 것이 깨졌을 때) 하나씩 패치들을 제거해가며 디버그 하기 쉽도록 만들어 준다.
- 2) 작은 패치들을 보내는 것뿐만 아니라 패치들을 제출하기전에 재작성하고 간단하게 (혹은 간단하게 재배치하여) 하는 것도 중요하다.

여기에 커널 개발자 Al Viro 의 이야기가 있다.

“학생의 수학 숙제를 채점하는 선생님을 생각해보라. 선생님은 학생들이 답을 얻을때까지 겪은 시행착오를 보길 원하지 않는다. 선생님들은 간결하고 가장 뛰어난 답을 보길 원한다. 훌륭한 학생은 이것을 알고 마지막으로 답을 얻기 전 중간 과정들을 제출하진 않는다.

커널 개발도 마찬가지이다. 메인테이너들과 검토하는 사람들은 문제를 풀어나가는 과정속에 숨겨진 과정을 보길 원하진 않는다. 그들은 간결하고 멋진 답을 보길 원한다.”

커뮤니티와 협력하며 뛰어난 답을 찾는 것과 여러분들의 끝마치지 못한 작업들 사이에 균형을 유지해야 하는 것은 어려울지도 모른다. 그러므로 프로세스의 초반에 여러분의 작업을 향상시키기위한 피드백을 얻는 것 뿐만 아니라 여러분들의 변경들을 작은 묶음으로 유지해서 심지어는 여러분의 작업의 모든 부분이 지금은 포함될 준비가 되어있지 않지만 작은 부분은 벌써 받아들여질 수 있도록 유지하는 것이 바람직하다.

또한 완성되지 않았고 “나중에 수정될 것이다.” 와 같은 것들을 포함하는 패치들은 받아들여지지 않을 것이라는 점을 유념하라.

## \* 변경을 정당화해라

여러분들의 나누어진 패치들을 리눅스 커뮤니티가 왜 반영해야 하는지를 알도록 하는 것은 매우 중요하다. 새로운 기능들이 필요하고 유용하다는 것은 반드시 그에 합당한 이유가 있어야 한다.

## \* 변경을 문서화해라

여러분이 패치를 보내려 할때는 여러분이 무엇을 말하려고 하는지를 충분히 생각하여 이메일을 작성해야 한다. 이 정보는 패치를 위한 ChangeLog 가 될 것이다. 그리고 항상 그 내용을 보길 원하는 모든 사람들을 위해 보존될 것이다. 패치는 완벽하게 다음과 같은 내용들을 포함하여 설명해야 한다.

- 변경이 왜 필요한지
- 패치에 관한 전체 설계 접근 (approach)
- 구현 상세들
- 테스트 결과들

이것이 무엇인지 더 자세한 것을 알고 싶다면 다음 문서의 ChageLog 항을 봐라.

“The Perfect Patch”

<https://www.ozlabs.org/~akpm/stuff/tpp.txt>

이 모든 것을 하는 것은 매우 어려운 일이다. 완벽히 소화하는 데는 적어도 몇년이 걸릴 수도 있다. 많은 인내와 결심이 필요한 계속되는 개선의 과정이다. 그러나 가능한한 포기하지 말라. 많은 사람들은 이전부터 해왔던 것이고 그 사람들도 정확하게 여러분들이 지금 서 있는 그 곳부터 시작했었다.

---

“개발 프로세스”(<https://lwn.net/Articles/94386/>) 섹션을 작성하는데 있어 참고할 문서를 사용하도록 허락해준 Paolo Ciarrocchi에게 감사한다. 여러분들이 말해야 할 것과 말해서는 안되는 것의 목록 중 일부를 제공해준 Randy Dunlap과 Gerrit Huizenga에게 감사한다. 또한 검토와 의견 그리고 공헌을 아끼지 않은 Pat Mochel, Hanna Linder, Randy Dunlap, Kay Sievers, Vojtech Pavlik, Jan Kara, Josh Boyer, Kees Cook, Andrew Morton, Andi Kleen, Vadim Lobanov, Jesper Juhl, Adrian Bunk, Keri Harris, Frans Pop, David A. Wheeler, Junio Hamano, Michael Kerrisk, and Alex Shepard에게도 감사를 전한다. 그들의 도움이 없었다면 이 문서는 존재하지 않았을 것이다.

메인테이너: Greg Kroah-Hartman <[greg@kroah.com](mailto:greg@kroah.com)>

## \* 리눅스 커널 메모리 배리어

NOTE:

This is a version of Documentation/memory-barriers.txt translated into Korean.

This document is maintained by SeongJae Park <[sj@kernel.org](mailto:sj@kernel.org)>.

If you find any difference between this document and the original file or a problem with the translation, please contact the maintainer of this file.

Please also note that the purpose of this file is to be easier to

read for non English (read: Korean) speakers and is not intended as a fork. So if you have any comments or updates for this file please update the original English file first. The English version is definitive, and readers should look there if they have any doubt.

=====

이 문서는  
Documentation/memory-barriers.txt  
의 한글 번역입니다.

역자 : 박성재 <sj@kernel.org>

=====

=====

리눅스 커널 메모리 배리어

=====

저자: David Howells <dhowells@redhat.com>  
Paul E. McKenney <paulmck@linux.ibm.com>  
Will Deacon <will.deacon@arm.com>  
Peter Zijlstra <peterz@infradead.org>

=====

면책조항

=====

이 문서는 명세서가 아닙니다; 이 문서는 완벽하지 않은데, 간결성을 위해 의도된 부분도 있고, 의도하지 않았지만 사람에 의해 쓰였다보니 불완전한 부분도 있습니다.  
이 문서는 리눅스에서 제공하는 다양한 메모리 배리어들을 사용하기 위한 안내서입니다만, 뭔가 이상하다 싶으면 (그런게 많을 겁니다) 질문을 부탁드립니다.  
일부 이상한 점들은 공식적인 메모리 일관성 모델과 tools/memory-model/에 있는 관련 문서를 참고해서 해결될 수 있을 겁니다. 그러나, 이 메모리 모델조차도 그 관리자들의 의견의 집합으로 봐야지, 절대 옳은 예언자로 신봉해선 안될 겁니다.

다시 말하지만, 이 문서는 리눅스가 하드웨어에 기대하는 사항에 대한 명세서가 아닙니다.

이 문서의 목적은 두 가지입니다:

- (1) 어떤 특정 배리어에 대해 기대할 수 있는 최소한의 기능을 명세하기 위해서,  
그리고
- (2) 사용 가능한 배리어들에 대해 어떻게 사용해야 하는지에 대한 안내를 제공하기  
위해서.

어떤 아키텍쳐는 특정한 배리어들에 대해서는 여기서 이야기하는 최소한의 요구사항들보다 많은 기능을 제공할 수도 있습니다만, 여기서 이야기하는 요구사항들을 충족하지 않는 아키텍쳐가 있다면 그 아키텍쳐가 잘못된 것이라고 알아두시기 바랍니다.

또한, 특정 아키텍쳐에서 일부 배리어는 해당 아키텍쳐의 특수한 동작 방식으로 인해

해당 배리어의 명시적 사용이 불필요해서 no-op 이 될수도 있음을 알아두시기 바랍니다.

역자: 본 번역 역시 완벽하지 않은데, 이 역시 부분적으로는 의도된 것이기도 합니다. 여타 기술 문서들이 그렇듯 완벽한 이해를 위해서는 번역문과 원문을 함께 읽으시되 번역문을 하나의 가이드로 활용하시길 추천드리며, 발견되는 오역 등에 대해서는 언제든 의견을 부탁드립니다. 과한 번역으로 인한 오해를 최소화하기 위해 애매한 부분이 있을 경우에는 어색함이 있더라도 원래의 용어를 차용합니다.

=====

목차:

=====

(\*) 추상 메모리 액세스 모델.

- 디바이스 오퍼레이션.
- 보장사항.

(\*) 메모리 배리어란 무엇인가?

- 메모리 배리어의 종류.
- 메모리 배리어에 대해 가정해선 안될 것.
- 데이터 의존성 배리어 (역사적).
- 컨트롤 의존성.
- SMP 배리어 짹맞추기.
- 메모리 배리어 시퀀스의 예.
- 읽기 메모리 배리어 vs 로드 예측.
- Multicopy 원자성.

(\*) 명시적 커널 배리어.

- 컴파일러 배리어.
- CPU 메모리 배리어.

(\*) 암묵적 커널 메모리 배리어.

- 락 Acquisition 함수.
- 인터럽트 비활성화 함수.
- 슬립과 웨이크업 함수.
- 그외의 함수들.

(\*) CPU 간 ACQUIRING 배리어의 효과.

- Acquire vs 메모리 액세스.

(\*) 메모리 배리어가 필요한 곳

- 프로세서간 상호 작용.
- 어토믹 오퍼레이션.
- 디바이스 액세스.
- 인터럽트.

(\*) 커널 I/O 배리어의 효과.

(\*) 가정되는 가장 완화된 실행 순서 모델.

(\*) CPU 캐시의 영향.

- 캐시 일관성.
- 캐시 일관성 vs DMA.
- 캐시 일관성 vs MMIO.

(\*) CPU 들이 저지르는 일들.

- 그리고, Alpha 가 있다.
- 가상 머신 게스트.

(\*) 사용 예.

- 순환식 버퍼.

(\*) 참고 문현.

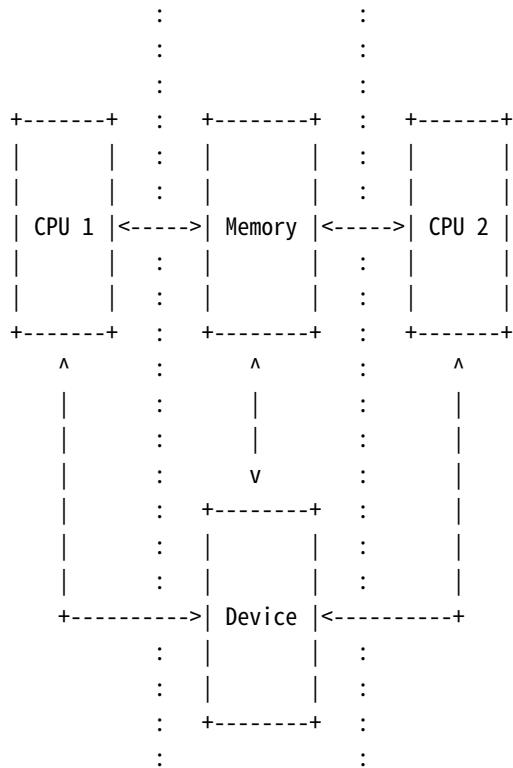
---

### =====

### 추상 메모리 액세스 모델

### =====

다음과 같이 추상화된 시스템 모델을 생각해 봅시다:



프로그램은 여러 메모리 액세스 오퍼레이션을 발생시키고, 각각의 CPU 는 그런 프로그램들을 실행합니다. 추상화된 CPU 모델에서 메모리 오퍼레이션들의 순서는 매우 완화되어 있고, CPU 는 프로그램이 인과관계를 어기지 않는 상태로 관리된다고 보일 수만 있다면 메모리 오퍼레이션을 자신이 원하는 어떤 순서대로든 재배치해 동작시킬 수 있습니다. 비슷하게, 컴파일러 또한 프로그램의 정상적 동작을 해치지 않는 한도 내에서는 어떤 순서로든 자신이 원하는 대로 인스트럭션을 재배치 할 수 있습니다.

따라서 위의 다이어그램에서 한 CPU가 동작시키는 메모리 오퍼레이션이 만들어내는 변화는 해당 오퍼레이션이 CPU 와 시스템의 다른 부분들 사이의 인터페이스(점선)를 지나가면서 시스템의 나머지 부분들에 인지됩니다.

예를 들어, 다음의 일련의 이벤트들을 생각해 봅시다:

CPU 1	CPU 2
=====	=====
{ A == 1; B == 2 }	
A = 3;	x = B;
B = 4;	y = A;

다이어그램의 가운데에 위치한 메모리 시스템에 보여지게 되는 액세스들은 다음의 총 24개의 조합으로 재구성될 수 있습니다:

STORE A=3,	STORE B=4,	y=LOAD A->3,	x=LOAD B->4
STORE A=3,	STORE B=4,	x=LOAD B->4,	y=LOAD A->3
STORE A=3,	y=LOAD A->3,	STORE B=4,	x=LOAD B->4
STORE A=3,	y=LOAD A->3,	x=LOAD B->2,	STORE B=4
STORE A=3,	x=LOAD B->2,	STORE B=4,	y=LOAD A->3
STORE A=3,	x=LOAD B->2,	y=LOAD A->3,	STORE B=4
STORE B=4,	STORE A=3,	y=LOAD A->3,	x=LOAD B->4
STORE B=4, ...			
...			

따라서 다음의 네가지 조합의 값들이 나올 수 있습니다:

```

x == 2, y == 1
x == 2, y == 3
x == 4, y == 1
x == 4, y == 3

```

한발 더 나아가서, 한 CPU 가 메모리 시스템에 반영한 스토어 오퍼레이션들의 결과는 다른 CPU 에서의 로드 오퍼레이션을 통해 인지되는데, 이 때 스토어가 반영된 순서와 다른 순서로 인지될 수도 있습니다.

예로, 아래의 일련의 이벤트들을 생각해 봅시다:

CPU 1	CPU 2
=====	=====
{ A == 1, B == 2, C == 3, P == &A, Q == &C }	

```
B = 4;           Q = P;  
P = &B;         D = *Q;
```

D로 읽혀지는 값은 CPU 2에서 P로부터 읽혀진 주소값에 의존적이기 때문에 여기엔 분명한 데이터 의존성이 있습니다. 하지만 이 이벤트들의 실행 결과로는 아래의 결과들이 모두 나타날 수 있습니다:

```
(Q == &A) and (D == 1)  
(Q == &B) and (D == 2)  
(Q == &B) and (D == 4)
```

CPU 2는 \*Q의 로드를 요청하기 전에 P를 Q에 넣기 때문에 D에 C를 집어넣는 일은 없음을 알아두세요.

### 디바이스 오퍼레이션

---

일부 디바이스는 자신의 컨트롤 인터페이스를 메모리의 특정 영역으로 매핑해서 제공하는데(Memory mapped I/O), 해당 컨트롤 레지스터에 접근하는 순서는 매우 중요합니다. 예를 들어, 어드레스 포트 레지스터 (A)와 데이터 포트 레지스터 (D)를 통해 접근되는 내부 레지스터 집합을 갖는 이더넷 카드를 생각해 봅시다. 내부의 5번 레지스터를 읽기 위해 다음의 코드가 사용될 수 있습니다:

```
*A = 5;  
x = *D;
```

하지만, 이건 다음의 두 조합 중 하나로 만들어질 수 있습니다:

```
STORE *A = 5, x = LOAD *D  
x = LOAD *D, STORE *A = 5
```

두번째 조합은 데이터를 읽어온 \_후에\_ 주소를 설정하므로, 오동작을 일으킬 겁니다.

### 보장사항

---

CPU에게 기대할 수 있는 최소한의 보장사항 몇 가지가 있습니다:

(\*) 어떤 CPU든, 의존성이 존재하는 메모리 액세스들은 해당 CPU 자신에게 있어서는 순서대로 메모리 시스템에 수행 요청됩니다. 즉, 다음에 대해서:

```
Q = READ_ONCE(P); D = READ_ONCE(*Q);
```

CPU는 다음과 같은 메모리 오퍼레이션 시퀀스를 수행 요청합니다:

```
Q = LOAD P, D = LOAD *Q
```

그리고 그 시퀀스 내에서의 순서는 항상 지켜집니다. 하지만, DEC Alpha에서 READ\_ONCE()는 메모리 배리어 명령도 내게 되어 있어서, DEC Alpha CPU는 다음과 같은 메모리 오퍼레이션들을 내놓게 됩니다:

`Q = LOAD P, MEMORY_BARRIER, D = LOAD *Q, MEMORY_BARRIER`

DEC Alpha에서 수행되든 아니든, `READ_ONCE()`는 컴파일러로부터의 악영향 또한 제거합니다.

(\*) 특정 CPU 내에서 겹치는 영역의 메모리에 행해지는 로드와 스토어 들은 해당 CPU 안에서는 순서가 바뀌지 않은 것으로 보여집니다. 즉, 다음에 대해서:

`a = READ_ONCE(*X); WRITE_ONCE(*X, b);`

CPU는 다음의 메모리 오퍼레이션 시퀀스만을 메모리에 요청할 겁니다:

`a = LOAD *X, STORE *X = b`

그리고 다음에 대해서는:

`WRITE_ONCE(*X, c); d = READ_ONCE(*X);`

CPU는 다음의 수행 요청만을 만들어냅니다:

`STORE *X = c, d = LOAD *X`

(로드 오퍼레이션과 스토어 오퍼레이션이 겹치는 메모리 영역에 대해 수행된다면 해당 오퍼레이션들은 겹친다고 표현됩니다).

그리고 반드시 또는 절대로 가정하거나 가정하지 말아야 하는 것들이 있습니다:

- (\*) 컴파일러가 `READ_ONCE()`나 `WRITE_ONCE()`로 보호되지 않은 메모리 액세스를 당신이 원하는 대로 할 것이라는 가정은 절대로 해선 안됩니다. 그것들이 없다면, 컴파일러는 컴파일러 배리어 섹션에서 다루게 될, 모든 "창의적인" 변경들을 만들어낼 권한을 갖게 됩니다.
- (\*) 개별적인 로드와 스토어들이 주어진 순서대로 요청될 것이라는 가정은 절대로 하지 말아야 합니다. 이 말은 곧:

`X = *A; Y = *B; *D = Z;`

는 다음의 것들 중 어느 것으로든 만들어질 수 있다는 의미입니다:

```
X = LOAD *A, Y = LOAD *B, STORE *D = Z
X = LOAD *A, STORE *D = Z, Y = LOAD *B
Y = LOAD *B, X = LOAD *A, STORE *D = Z
Y = LOAD *B, STORE *D = Z, X = LOAD *A
STORE *D = Z, X = LOAD *A, Y = LOAD *B
STORE *D = Z, Y = LOAD *B, X = LOAD *A
```

- (\*) 겹치는 메모리 액세스들은 합쳐지거나 버려질 수 있음을 반드시 가정해야 합니다. 다음의 코드는:

`X = *A; Y = *(A + 4);`

다음의 것들 중 뭐든 될 수 있습니다:

```
X = LOAD *A; Y = LOAD *(A + 4);  
Y = LOAD *(A + 4); X = LOAD *A;  
{X, Y} = LOAD {*A, *(A + 4)};
```

그리고:

```
*A = X; *(A + 4) = Y;
```

는 다음 중 뭐든 될 수 있습니다:

```
STORE *A = X; STORE *(A + 4) = Y;  
STORE *(A + 4) = Y; STORE *A = X;  
STORE {*A, *(A + 4)} = {X, Y};
```

그리고 보장사항에 반대되는 것들(anti-guarantees)이 있습니다:

- (\*) 이 보장사항들은 bitfield 에는 적용되지 않는데, 컴파일러들은 bitfield 를 수정하는 코드를 생성할 때 원자성 없는(non-atomic) 읽고-수정하고-쓰는 인스트럭션들의 조합을 만드는 경우가 많기 때문입니다. 병렬 알고리즘의 동기화에 bitfield 를 사용하려 하지 마십시오.
- (\*) bitfield 들이 여러 락으로 보호되는 경우라 하더라도, 하나의 bitfield 의 모든 필드들은 하나의 락으로 보호되어야 합니다. 만약 한 bitfield 의 두 필드가 서로 다른 락으로 보호된다면, 컴파일러의 원자성 없는 읽고-수정하고-쓰는 인스트럭션 조합은 한 필드에의 업데이트가 근처의 필드에도 영향을 끼치게 할 수 있습니다.
- (\*) 이 보장사항들은 적절하게 정렬되고 크기가 잡힌 스칼라 변수들에 대해서만 적용됩니다. "적절하게 크기가 잡힌" 이라함은 현재로써는 "char", "short", "int" 그리고 "long" 과 같은 크기의 변수들을 의미합니다. "적절하게 정렬된"은 자연스런 정렬을 의미하는데, 따라서 "char" 에 대해서는 아무 제약이 없고, "short" 에 대해서는 2바이트 정렬을, "int" 에는 4바이트 정렬을, 그리고 "long" 에 대해서는 32-bit 시스템인지 64-bit 시스템인지에 따라 4바이트 또는 8바이트 정렬을 의미합니다. 이 보장사항들은 C11 표준에서 소개되었으므로, C11 전의 오래된 컴파일러(예를 들어, gcc 4.6) 를 사용할 때엔 주의하시기 바랍니다. 표준에 이 보장사항들은 "memory location" 을 정의하는 3.14 섹션에 다음과 같이 설명되어 있습니다:  
(역자: 인용문이므로 번역하지 않습니다)

**memory location**  
either an object of scalar type, or a maximal sequence  
of adjacent bit-fields all having nonzero width

NOTE 1: Two threads of execution can update and access  
separate memory locations without interfering with  
each other.

NOTE 2: A bit-field and an adjacent non-bit-field member  
are in separate memory locations. The same applies  
to two bit-fields, if one is declared inside a nested

structure declaration and the other is not, or if the two are separated by a zero-length bit-field declaration, or if they are separated by a non-bit-field member declaration. It is not safe to concurrently update two bit-fields in the same structure if all members declared between them are also bit-fields, no matter what the sizes of those intervening bit-fields happen to be.

---

### ===== 메모리 배리어란 무엇인가? =====

앞에서 봤듯이, 상호간 의존성이 없는 메모리 오퍼레이션들은 실제로는 무작위적 순서로 수행될 수 있으며, 이는 CPU 와 CPU 간의 상호작용이나 I/O 에 문제가 될 수 있습니다. 따라서 컴파일러와 CPU 가 순서를 바꾸는데 제약을 걸 수 있도록 개입할 수 있는 어떤 방법이 필요합니다.

메모리 배리어는 그런 개입 수단입니다. 메모리 배리어는 배리어를 사이에 둔 앞과 뒤 양측의 메모리 오퍼레이션들 간에 부분적 순서가 존재하도록 하는 효과를 줍니다.

시스템의 CPU 들과 여러 디바이스들은 성능을 올리기 위해 명령어 재배치, 실행 유예, 메모리 오퍼레이션들의 조합, 예측적 로드(speculative load), 브랜치 예측(speculative branch prediction), 다양한 종류의 캐싱(caching) 등의 다양한 트릭을 사용할 수 있기 때문에 이런 강제력은 중요합니다. 메모리 배리어들은 이런 트릭들을 무효로 하거나 억제하는 목적으로 사용되어져서 코드가 여러 CPU 와 디바이스들 간의 상호작용을 정상적으로 제어할 수 있게 해줍니다.

---

### 메모리 배리어의 종류

---

메모리 배리어는 네개의 기본 타입으로 분류됩니다:

- (1) 쓰기 (또는 스토어) 메모리 배리어.

쓰기 메모리 배리어는 시스템의 다른 컴포넌트들에 해당 배리어보다 앞서 명시된 모든 STORE 오퍼레이션들이 해당 배리어 뒤에 명시된 모든 STORE 오퍼레이션들보다 먼저 수행된 것으로 보일 것을 보장합니다.

쓰기 배리어는 스토어 오퍼레이션들에 대한 부분적 순서 세우기입니다; 로드 오퍼레이션들에 대해서는 어떤 영향도 끼치지 않습니다.

CPU 는 시간의 흐름에 따라 메모리 시스템에 일련의 스토어 오퍼레이션들을 하나씩 요청해 집어넣습니다. 쓰기 배리어 앞의 모든 스토어 오퍼레이션들은 쓰기 배리어 뒤의 모든 스토어 오퍼레이션들보다 앞서 수행될 겁니다.

[!] 쓰기 배리어들은 읽기 또는 데이터 의존성 배리어와 함께 짹을 맞춰 사용되어야만 함을 알아두세요; "SMP 배리어 짹맞추기" 서브섹션을 참고하세요.

- (2) 데이터 의존성 배리어.

데이터 의존성 배리어는 읽기 배리어의 보다 완화된 형태입니다. 두개의 로드 오퍼레이션이 있고 두번째 것이 첫번째 것의 결과에 의존하고 있을 때(예: 두번째 로드가 참조할 주소를 첫번째 로드가 읽는 경우), 두번째 로드가 읽어올 데이터는 첫번째 로드에 의해 그 주소가 얻어진 뒤에 업데이트 됨을 보장하기 위해서 데이터 의존성 배리어가 필요할 수 있습니다.

데이터 의존성 배리어는 상호 의존적인 로드 오퍼레이션들 사이의 부분적 순서 세우기입니다; 스토어 오퍼레이션들이나 독립적인 로드들, 또는 중복되는 로드들에 대해서는 어떤 영향도 끼치지 않습니다.

(1) 에서 언급했듯이, 시스템의 CPU들은 메모리 시스템에 일련의 스토어 오퍼레이션들을 던져 넣고 있으며, 거기에 관심이 있는 다른 CPU는 그 오퍼레이션들을 메모리 시스템이 실행한 결과를 인지할 수 있습니다. 이처럼 다른 CPU의 스토어 오퍼레이션의 결과에 관심을 두고 있는 CPU가 수행 요청한 데이터 의존성 배리어는, 배리어 앞의 어떤 로드 오퍼레이션이 다른 CPU에서 던져 넣은 스토어 오퍼레이션과 같은 영역을 향했다면, 그런 스토어 오퍼레이션들이 만들어내는 결과가 데이터 의존성 배리어 뒤의 로드 오퍼레이션들에게는 보일 것을 보장합니다.

이 순서 세우기 제약에 대한 그림을 보기 위해선 "메모리 배리어 시퀀스의 예" 서브섹션을 참고하시기 바랍니다.

[!] 첫번째 로드는 반드시 \_데이터\_ 의존성을 가져야지 컨트롤 의존성을 가져야 하는게 아님을 알아두십시오. 만약 두번째 로드를 위한 주소가 첫번째 로드에 의존적이지만 그 의존성은 조건적이지 그 주소 자체를 가져오는게 아니라면, 그것은 컨트롤\_ 의존성이고, 이 경우에는 읽기 배리어나 그보다 강력한 무언가가 필요합니다. 더 자세한 내용을 위해서는 "컨트롤 의존성" 서브섹션을 참고하시기 바랍니다.

[!] 데이터 의존성 배리어는 보통 쓰기 배리어들과 함께 짹을 맞춰 사용되어야 합니다; "SMP 배리어 짹맞추기" 서브섹션을 참고하세요.

### (3) 읽기 (또는 로드) 메모리 배리어.

읽기 배리어는 데이터 의존성 배리어 기능의 보장사항에 더해서 배리어보다 앞서 명시된 모든 LOAD 오퍼레이션들이 배리어 뒤에 명시되는 모든 LOAD 오퍼레이션들보다 먼저 행해진 것으로 시스템의 다른 컴포넌트들에 보여질 것을 보장합니다.

읽기 배리어는 로드 오퍼레이션에 행해지는 부분적 순서 세우기입니다; 스토어 오퍼레이션에 대해서는 어떤 영향도 끼치지 않습니다.

읽기 메모리 배리어는 데이터 의존성 배리어를 내장하므로 데이터 의존성 배리어를 대신할 수 있습니다.

[!] 읽기 배리어는 일반적으로 쓰기 배리어들과 함께 짹을 맞춰 사용되어야 합니다; "SMP 배리어 짹맞추기" 서브섹션을 참고하세요.

### (4) 범용 메모리 배리어.

범용(general) 메모리 배리어는 배리어보다 앞서 명시된 모든 LOAD 와 STORE  
오퍼레이션들이 배리어 뒤에 명시된 모든 LOAD 와 STORE 오퍼레이션들보다  
먼저 수행된 것으로 시스템의 나머지 컴포넌트들에 보이게 됨을 보장합니다.

범용 메모리 배리어는 로드와 스토어 모두에 대한 부분적 순서 세우기입니다.

범용 메모리 배리어는 읽기 메모리 배리어, 쓰기 메모리 배리어 모두를  
내장하므로, 두 배리어를 모두 대신할 수 있습니다.

그리고 두개의 명시적이지 않은 타입이 있습니다:

#### (5) ACQUIRE 오퍼레이션.

이 타입의 오퍼레이션은 단방향의 투과성 배리어처럼 동작합니다. ACQUIRE  
오퍼레이션 뒤의 모든 메모리 오퍼레이션들이 ACQUIRE 오퍼레이션 후에  
일어난 것으로 시스템의 나머지 컴포넌트들에 보이게 될 것이 보장됩니다.  
LOCK 오퍼레이션과 smp\_load\_acquire(), smp\_cond\_load\_acquire() 오퍼레이션도  
ACQUIRE 오퍼레이션에 포함됩니다.

ACQUIRE 오퍼레이션 앞의 메모리 오퍼레이션들은 ACQUIRE 오퍼레이션 완료 후에  
수행된 것처럼 보일 수 있습니다.

ACQUIRE 오퍼레이션은 거의 항상 RELEASE 오퍼레이션과 짹을 지어 사용되어야  
합니다.

#### (6) RELEASE 오퍼레이션.

이 타입의 오퍼레이션들도 단방향 투과성 배리어처럼 동작합니다. RELEASE  
오퍼레이션 앞의 모든 메모리 오퍼레이션들은 RELEASE 오퍼레이션 전에 완료된  
것으로 시스템의 다른 컴포넌트들에 보여질 것이 보장됩니다. UNLOCK 류의  
오퍼레이션들과 smp\_store\_release() 오퍼레이션도 RELEASE 오퍼레이션의  
일종입니다.

RELEASE 오퍼레이션 뒤의 메모리 오퍼레이션들은 RELEASE 오퍼레이션이  
완료되기 전에 행해진 것처럼 보일 수 있습니다.

ACQUIRE 와 RELEASE 오퍼레이션의 사용은 일반적으로 다른 메모리 배리어의  
필요성을 없앱니다. 또한, RELEASE+ACQUIRE 조합은 범용 메모리 배리어처럼  
동작할 것을 보장하지 -않습니다-. 하지만, 어떤 변수에 대한 RELEASE  
오퍼레이션을 앞서는 메모리 액세스들의 수행 결과는 이 RELEASE 오퍼레이션을  
뒤이어 같은 변수에 대해 수행된 ACQUIRE 오퍼레이션을 뒤따르는 메모리  
액세스에는 보여질 것이 보장됩니다. 다르게 말하자면, 주어진 변수의  
크리티컬 섹션에서는, 해당 변수에 대한 앞의 크리티컬 섹션에서의 모든  
액세스들이 완료되었을 것을 보장합니다.

즉, ACQUIRE 는 최소한의 "취득" 동작처럼, 그리고 RELEASE 는 최소한의 "공개"  
처럼 동작한다는 의미입니다.

atomic\_t.txt 에 설명된 어토믹 오퍼레이션들 중 일부는 완전히 순서잡힌 것들과

(배리어를 사용하지 않는) 완화된 순서의 것들 외에 ACQUIRE 와 RELEASE 부류의 것들도 존재합니다. 로드와 스토어를 모두 수행하는 조합된 어토믹 오퍼레이션에서, ACQUIRE 는 해당 오퍼레이션의 로드 부분에만 적용되고 RELEASE 는 해당 오퍼레이션의 스토어 부분에만 적용됩니다.

메모리 배리어들은 두 CPU 간, 또는 CPU 와 디바이스 간에 상호작용의 가능성이 있을 때에만 필요합니다. 만약 어떤 코드에 그런 상호작용이 없을 것이 보장된다면, 해당 코드에서는 메모리 배리어를 사용할 필요가 없습니다.

이것들은 최소한의 보장사항들임을 알아두세요. 다른 아키텍쳐에서는 더 강력한 보장사항을 제공할 수도 있습니다만, 그런 보장사항은 아키텍쳐 종속적 코드 이외의 부분에서는 신뢰되지 않을 겁니다.

### 메모리 배리어에 대해 가정해선 안될 것

---

리눅스 커널 메모리 배리어들이 보장하지 않는 것들이 있습니다:

- (\*) 메모리 배리어 앞에서 명시된 어떤 메모리 액세스도 메모리 배리어 명령의 수행 완료 시점까지 완료 될 것이란 보장은 없습니다; 배리어가 하는 일은 CPU 의 액세스 큐에 특정 타입의 액세스들은 넘을 수 없는 선을 긋는 것으로 생각될 수 있습니다.
- (\*) 한 CPU 에서 메모리 배리어를 수행하는게 시스템의 다른 CPU 나 하드웨어에 어떤 직접적인 영향을 끼친다는 보장은 존재하지 않습니다. 배리어 수행이 만드는 간접적 영향은 두번째 CPU 가 첫번째 CPU 의 액세스들의 결과를 바라보는 순서가 됩니다만, 다음 항목을 보세요:
- (\*) 첫번째 CPU 가 두번째 CPU 의 메모리 액세스들의 결과를 바라볼 때, 설령 두번째 CPU 가 메모리 배리어를 사용한다 해도, 첫번째 CPU 또한 그에 맞는 메모리 배리어를 사용하지 않는다면 ("SMP 배리어 짹맞추기" 서브섹션을 참고하세요) 그 결과가 올바른 순서로 보여진다는 보장은 없습니다.
- (\*) CPU 바깥의 하드웨어[\*] 가 메모리 액세스들의 순서를 바꾸지 않는다는 보장은 존재하지 않습니다. CPU 캐시 일관성 메커니즘은 메모리 배리어의 간접적 영향을 CPU 사이에 전파하긴 하지만, 순서대로 전파하지는 않을 수 있습니다.

[\*] 버스 마스터링 DMA 와 일관성에 대해서는 다음을 참고하시기 바랍니다:

[Documentation/driver-api/pci/pci.rst](#)  
[Documentation/core-api/dma-api-howto.rst](#)  
[Documentation/core-api/dma-api.rst](#)

### 데이터 의존성 배리어 (역사적)

---

리눅스 커널 v4.15 기준으로, `smp_mb()` 가 DEC Alpha 용 `READ_ONCE()` 코드에 추가되었는데, 이는 이 섹션에 주의를 기울여야 하는 사람들은 DEC Alpha 아키텍쳐 전용 코드를 만드는 사람들과 `READ_ONCE()` 자체를 만드는 사람들 뿐임을 의미합니다.

그런 분들을 위해, 그리고 역사에 관심 있는 분들을 위해, 여기 데이터 의존성 배리어에 대한 이야기를 적습니다.

데이터 의존성 배리어의 사용에 있어 지켜야 하는 사항들은 약간 미묘하고, 데이터 의존성 배리어가 사용되어야 하는 상황도 항상 명백하지는 않습니다. 설명을 위해 다음의 이벤트 시퀀스를 생각해 봅시다:

CPU 1	CPU 2
=====	=====
{ A == 1, B == 2, C == 3, P == &A, Q == &C }	
B = 4;	
<쓰기 배리어>	
WRITE_ONCE(P, &B)	
Q = READ_ONCE(P);	
D = *Q;	

여기엔 분명한 데이터 의존성이 존재하므로, 이 시퀀스가 끝났을 때 Q 는 &A 또는 &B 일 것이고, 따라서:

(Q == &A) 는 (D == 1) 를,  
(Q == &B) 는 (D == 4) 를 의미합니다.

하지만! CPU 2 는 B 의 업데이트를 인식하기 전에 P 의 업데이트를 인식할 수 있고, 따라서 다음의 결과가 가능합니다:

(Q == &B) and (D == 2) ????

이런 결과는 일관성이나 인과 관계 유지가 실패한 것처럼 보일 수도 있겠지만, 그렇지 않습니다, 그리고 이 현상은 (DEC Alpha 와 같은) 여러 CPU 에서 실제로 발견될 수 있습니다.

이 문제 상황을 제대로 해결하기 위해, 데이터 의존성 배리어나 그보다 강화된 무언가가 주소를 읽어올 때와 데이터를 읽어올 때 사이에 추가되어야만 합니다:

CPU 1	CPU 2
=====	=====
{ A == 1, B == 2, C == 3, P == &A, Q == &C }	
B = 4;	
<쓰기 배리어>	
WRITE_ONCE(P, &B);	
Q = READ_ONCE(P);	
<데이터 의존성 배리어>	
D = *Q;	

이 변경은 앞의 처음 두가지 결과 중 하나만이 발생할 수 있고, 세번째의 결과는 발생할 수 없도록 합니다.

[!] 이 상당히 반직관적인 상황은 분리된 캐시를 가지는 기계들에서 가장 잘 발생하는데, 예를 들면 한 캐시 뱅크는 짹수 번호의 캐시 라인들을 처리하고, 다른 뱅크는 홀수 번호의 캐시 라인들을 처리하는 경우임을 알아두시기 바랍니다. 포인터 P 는 짹수 번호 캐시 라인에 저장되어 있고, 변수 B 는 홀수 번호 캐시 라인에

저장되어 있을 수 있습니다. 여기서 값을 읽어오는 CPU 의 캐시의 홀수 번호 처리 뱅크는 열심히 일감을 처리중인 반면 홀수 번호 처리 뱅크는 할 일 없이 한가한 중이라면 포인터 P (&B) 의 새로운 값과 변수 B 의 기존 값 (2) 를 볼 수 있습니다.

의존적 쓰기들의 순서를 맞추는데에는 데이터 의존성 배리어가 필요치 않은데, 이는 리눅스 커널이 지원하는 CPU 들은 (1) 쓰기가 정말로 일어날지, (2) 쓰기가 어디에 이루어질지, 그리고 (3) 쓰여질 값을 확실히 알기 전까지는 쓰기를 수행하지 않기 때문입니다. 하지만 "컨트롤 의존성" 섹션과

[Documentation/RCU/rcu\\_dereference.rst](#) 파일을 주의 깊게 읽어 주시기 바랍니다: 컴파일러는 매우 창의적인 많은 방법으로 종속성을 깨 수 있습니다.

CPU 1	CPU 2
=====	=====
{ A == 1, B == 2, C = 3, P == &A, Q == &C }	
B = 4;	
<쓰기 배리어>	
WRITE_ONCE(P, &B);	
	Q = READ_ONCE(P);
	WRITE_ONCE(*Q, 5);

따라서, Q 로의 읽기와 \*Q 로의 쓰기 사이에는 데이터 종속성 배리어가 필요치 않습니다. 달리 말하면, 데이터 종속성 배리어가 없더라도 다음 결과는 생기지 않습니다:

(Q == &B) && (B == 4)

이런 패턴은 드물게 사용되어야 함을 알아 두시기 바랍니다. 무엇보다도, 의존성 순서 규칙의 의도는 쓰기 작업을 -예방- 해서 그로 인해 발생하는 비싼 캐시 미스도 없애려는 것입니다. 이 패턴은 드물게 발생하는 에러 조건 같은것들을 기록하는데 사용될 수 있으며, CPU의 자연적인 순서 보장이 그런 기록들을 사라지지 않게 해줍니다.

데이터 의존성에 의해 제공되는 이 순서규칙은 이를 포함하고 있는 CPU 에 지역적임을 알아두시기 바랍니다. 더 많은 정보를 위해선 "Multicopy 원자성" 섹션을 참고하세요.

데이터 의존성 배리어는 매우 중요한데, 예를 들어 RCU 시스템에서 그렇습니다. `include/linux/rcupdate.h` 의 `rcu_assign_pointer()` 와 `rcu_dereference()` 를 참고하세요. 여기서 데이터 의존성 배리어는 RCU 로 관리되는 포인터의 타겟을 현재 타겟에서 수정된 새로운 타겟으로 바꾸는 작업에서 새로 수정된 타겟이 초기화가 완료되지 않은 채로 보여지는 일이 일어나지 않게 해줍니다.

더 많은 예를 위해선 "캐시 일관성" 서브섹션을 참고하세요.

컨트롤 의존성

---

현재의 컴파일러들은 컨트롤 의존성을 이해하고 있지 않기 때문에 컨트롤 의존성은

약간 다루기 어려울 수 있습니다. 이 섹션의 목적은 여러분이 컴파일러의 무시로 인해 여러분의 코드가 망가지는 걸 막을 수 있도록 돕는겁니다.

로드-로드 컨트롤 의존성은 데이터 의존성 배리어만으로는 정확히 동작할 수가 없어서 읽기 메모리 배리어를 필요로 합니다. 아래의 코드를 봅시다:

```
q = READ_ONCE(a);
if (q) {
    <데이터 의존성 배리어> /* BUG: No data dependency!!! */
    p = READ_ONCE(b);
}
```

이 코드는 원하는 대로의 효과를 내지 못할 수 있는데, 이 코드에는 데이터 의존성이 아니라 컨트롤 의존성이 존재하기 때문으로, 이런 상황에서 CPU는 실행 속도를 더 빠르게 하기 위해 분기 조건의 결과를 예측하고 코드를 재배치 할 수 있어서 다른 CPU는 b로부터의 로드 오퍼레이션이 a로부터의 로드 오퍼레이션보다 먼저 발생한 걸로 인식할 수 있습니다. 여기에 정말로 필요했던 건 다음과 같습니다:

```
q = READ_ONCE(a);
if (q) {
    <읽기 배리어>
    p = READ_ONCE(b);
}
```

하지만, 스토어 오퍼레이션은 예측적으로 수행되지 않습니다. 즉, 다음 예에서와 같이 로드-스토어 컨트롤 의존성이 존재하는 경우에는 순서가 -지켜진다-는 의미입니다.

```
q = READ_ONCE(a);
if (q) {
    WRITE_ONCE(b, 1);
}
```

컨트롤 의존성은 보통 다른 타입의 배리어들과 짹을 맞춰 사용됩니다. 그렇다면 하나, READ\_ONCE() 도 WRITE\_ONCE() 도 선택사항이 아니라 필수사항임을 부디 명심하세요! READ\_ONCE() 가 없다면, 컴파일러는 'a'로부터의 로드를 'a'로부터의 또 다른 로드와 조합할 수 있습니다. WRITE\_ONCE() 가 없다면, 컴파일러는 'b'로의 스토어를 'b'로의 또 라느 스토어들과 조합할 수 있습니다. 두 경우 모두 순서에 있어 상당히 비직관적인 결과를 초래할 수 있습니다.

이걸로 끝이 아닌게, 컴파일러가 변수 'a'의 값이 항상 0이 아니라고 증명할 수 있다면, 앞의 예에서 "if" 문을 없애서 다음과 같이 최적화 할 수도 있습니다:

```
q = a;
b = 1; /* BUG: Compiler and CPU can both reorder!!! */
```

그러니 READ\_ONCE() 를 반드시 사용하세요.

다음과 같이 "if" 문의 양갈래 브랜치에 모두 존재하는 동일한 스토어에 대해 순서를 강제하고 싶은 경우가 있을 수 있습니다:

```
q = READ_ONCE(a);
```

```
if (q) {
    barrier();
    WRITE_ONCE(b, 1);
    do_something();
} else {
    barrier();
    WRITE_ONCE(b, 1);
    do_something_else();
}
```

안타깝게도, 현재의 컴파일러들은 높은 최적화 레벨에서는 이걸 다음과 같이 바꿔버립니다:

```
q = READ_ONCE(a);
barrier();
WRITE_ONCE(b, 1); /* BUG: No ordering vs. load from a!!! */
if (q) {
    /* WRITE_ONCE(b, 1); -- moved up, BUG!!! */
    do_something();
} else {
    /* WRITE_ONCE(b, 1); -- moved up, BUG!!! */
    do_something_else();
}
```

이제 'a'에서의 로드와 'b'로의 스토어 사이에는 조건적 관계가 없기 때문에 CPU는 이들의 순서를 바꿀 수 있게 됩니다: 이런 경우에 조건적 관계는 반드시 필요한데, 모든 컴파일러 최적화가 이루어지고 난 후의 어셈블리 코드에서도 마찬가지입니다. 따라서, 이 예에서 순서를 지키기 위해서는 `smp_store_release()`와 같은 명시적 메모리 배리어가 필요합니다:

```
q = READ_ONCE(a);
if (q) {
    smp_store_release(&b, 1);
    do_something();
} else {
    smp_store_release(&b, 1);
    do_something_else();
}
```

반면에 명시적 메모리 배리어가 없다면, 이런 경우의 순서는 스토어 오퍼레이션들이 서로 다를 때에만 보장되는데, 예를 들면 다음과 같은 경우입니다:

```
q = READ_ONCE(a);
if (q) {
    WRITE_ONCE(b, 1);
    do_something();
} else {
    WRITE_ONCE(b, 2);
    do_something_else();
}
```

처음의 `READ_ONCE()`는 컴파일러가 'a'의 값을 증명해내는 것을 막기 위해 여전히

필요합니다.

또한, 로컬 변수 'q' 를 가지고 하는 일에 대해 주의해야 하는데, 그러지 않으면 컴파일러는 그 값을 추측하고 또다시 필요한 조건관계를 없애버릴 수 있습니다. 예를 들면:

```
q = READ_ONCE(a);
if (q % MAX) {
    WRITE_ONCE(b, 1);
    do_something();
} else {
    WRITE_ONCE(b, 2);
    do_something_else();
}
```

만약 MAX 가 1 로 정의된 상수라면, 컴파일러는 ( $q \% \text{MAX}$ ) 는 0이란 것을 알아채고, 위의 코드를 아래와 같이 바꿔버릴 수 있습니다:

```
q = READ_ONCE(a);
WRITE_ONCE(b, 2);
do_something_else();
```

이렇게 되면, CPU 는 변수 'a' 로부터의 로드와 변수 'b' 로의 스토어 사이의 순서를 지켜줄 필요가 없어집니다. `barrier()` 를 추가해 해결해 보고 싶겠지만, 그건 도움이 안됩니다. 조건 관계는 사라졌고, `barrier()` 는 이를 되돌리지 못합니다. 따라서, 이 순서를 지켜야 한다면, MAX 가 1 보다 크다는 것을, 다음과 같은 방법을 사용해 분명히 해야 합니다:

```
q = READ_ONCE(a);
BUILD_BUG_ON(MAX <= 1); /* Order load from a with store to b. */
if (q % MAX) {
    WRITE_ONCE(b, 1);
    do_something();
} else {
    WRITE_ONCE(b, 2);
    do_something_else();
}
```

'b' 로의 스토어들은 여전히 서로 다른 값을 알아두세요. 만약 그것들이 동일하면, 앞에서 이야기했듯, 컴파일러가 그 스토어 오퍼레이션들을 'if' 문 바깥으로 끄집어낼 수 있습니다.

또한 이진 조건문 평가에 너무 의존하지 않도록 조심해야 합니다. 다음의 예를 볼시다:

```
q = READ_ONCE(a);
if (q || 1 > 0)
    WRITE_ONCE(b, 1);
```

첫번째 조건만으로는 브랜치 조건 전체를 거짓으로 만들 수 없고 두번째 조건은 항상 참이기 때문에, 컴파일러는 이 예를 다음과 같이 바꿔서 컨트롤 의존성을 없애버릴 수 있습니다:

```
q = READ_ONCE(a);
WRITE_ONCE(b, 1);
```

이 예는 컴파일러가 코드를 추측으로 수정할 수 없도록 분명히 해야 한다는 점을 강조합니다. 조금 더 일반적으로 말해서, READ\_ONCE() 는 컴파일러에게 주어진 로드 오퍼레이션을 위한 코드를 정말로 만들도록 하지만, 컴파일러가 그렇게 만들어진 코드의 수행 결과를 사용하도록 강제하지는 않습니다.

또한, 컨트롤 의존성은 if 문의 then 절과 else 절에 대해서만 적용됩니다. 상세히 말해서, 컨트롤 의존성은 if 문을 뒤따르는 코드에는 적용되지 않습니다:

```
q = READ_ONCE(a);
if (q) {
    WRITE_ONCE(b, 1);
} else {
    WRITE_ONCE(b, 2);
}
WRITE_ONCE(c, 1); /* BUG: No ordering against the read from 'a'. */
```

컴파일러는 volatile 타입에 대한 액세스를 재배치 할 수 없고 이 조건 하의 'b' 로의 쓰기를 재배치 할 수 없기 때문에 순서 규칙이 존재한다고 주장하고 싶을 겁니다. 불행히도 이 경우에, 컴파일러는 다음의 가상의 pseudo-assembly 언어 코드처럼 'b' 로의 두개의 쓰기 오퍼레이션을 conditional-move 인스트럭션으로 번역할 수 있습니다:

```
ld r1,a
cmp r1,$0
cmov,ne r4,$1
cmov,eq r4,$2
st r4,b
st $1,c
```

완화된 순서 규칙의 CPU 는 'a'로부터의 로드와 'c' 로의 스토어 사이에 어떤 종류의 의존성도 갖지 않을 겁니다. 이 컨트롤 의존성은 두개의 cmov 인스트럭션과 거기에 의존하는 스토어에게만 적용될 겁니다. 짧게 말하자면, 컨트롤 의존성은 주어진 if 문의 then 절과 else 절에게만 (그리고 이 두 절 내에서 호출되는 함수들에게까지) 적용되지, 이 if 문을 뒤따르는 코드에는 적용되지 않습니다.

컨트롤 의존성에 의해 제공되는 이 순서규칙은 이를 포함하고 있는 CPU 에 지역적입니다. 더 많은 정보를 위해선 "Multicopy 원자성" 섹션을 참고하세요.

요약하자면:

(\*) 컨트롤 의존성은 앞의 로드들을 뒤의 스토어들에 대해 순서를 맞춰줍니다.  
하지만, 그 외의 어떤 순서도 보장하지 -않습니다-: 앞의 로드와 뒤의 로드들 사이에도, 앞의 스토어와 뒤의 스토어들 사이에도요. 이런 다른 형태의 순서가 필요하다면 smp\_rmb() 나 smp\_wmb()를, 또는, 앞의 스토어들과 뒤의 로드들 사이의 순서를 위해서는 smp\_mb() 를 사용하세요.

(\*) "if" 문의 양갈래 브랜치가 같은 변수에의 동일한 스토어로 시작한다면, 그 스토어들은 각 스토어 앞에 smp\_mb() 를 넣거나 smp\_store\_release() 를 사용해서 스토어를 하는 식으로 순서를 맞춰줘야 합니다. 이 문제를 해결하기 위해 "if" 문의 양갈래 브랜치의 시작 지점에 barrier() 를 넣는 것만으로는 충분한 해결이 되지 않는데, 이는 앞의 예에서 본것과 같이, 컴파일러의 최적화는 barrier() 가 의미하는 바를 지키면서도 컨트롤 의존성을 손상시킬 수 있기 때문이라는 점을 부디 알아두시기 바랍니다.

(\*) 컨트롤 의존성은 앞의 로드와 뒤의 스토어 사이에 최소 하나의, 실행 시점에서의 조건관계를 필요로 하며, 이 조건관계는 앞의 로드와 관계되어야 합니다. 만약 컴파일러가 조건 관계를 최적화로 없앨수 있다면, 순서도 최적화로 없애버렸을 겁니다. READ\_ONCE() 와 WRITE\_ONCE() 의 주의 깊은 사용은 주어진 조건 관계를 유지하는데 도움이 될 수 있습니다.

(\*) 컨트롤 의존성을 위해선 컴파일러가 조건관계를 없애버리는 것을 막아야 합니다. 주의 깊은 READ\_ONCE() 나 atomic{,64}\_read() 의 사용이 컨트롤 의존성이 사라지지 않게 하는데 도움을 줄 수 있습니다. 더 많은 정보를 위해선 "컴파일러 배리어" 섹션을 참고하시기 바랍니다.

(\*) 컨트롤 의존성은 컨트롤 의존성을 갖는 if 문의 then 절과 else 절과 이 두 절 내에서 호출되는 함수들에만 적용됩니다. 컨트롤 의존성은 컨트롤 의존성을 갖는 if 문을 뒤따르는 코드에는 적용되지 -않습니다-.

(\*) 컨트롤 의존성은 보통 다른 타입의 배리어들과 짹을 맞춰 사용됩니다.

(\*) 컨트롤 의존성은 multicopy 원자성을 제공하지 -않습니다-. 모든 CPU 들이 특정 스토어를 동시에 보길 원한다면, smp\_mb() 를 사용하세요.

(\*) 컴파일러는 컨트롤 의존성을 이해하고 있지 않습니다. 따라서 컴파일러가 여러분의 코드를 망가뜨리지 않도록 하는건 여러분이 해야 하는 일입니다.

## SMP 배리어 짹맞추기

---

CPU 간 상호작용을 다룰 때에 일부 타입의 메모리 배리어는 항상 짹을 맞춰 사용되어야 합니다. 적절하게 짹을 맞추지 않은 코드는 사실상 에러에 가깝습니다.

범용 배리어들은 범용 배리어끼리도 짹을 맞추지만 multicopy 원자성이 없는 대부분의 다른 타입의 배리어들과도 짹을 맞춥니다. ACQUIRE 배리어는 RELEASE 배리어와 짹을 맞춥니다만, 둘 다 범용 배리어를 포함해 다른 배리어들과도 짹을 맞출 수 있습니다. 쓰기 배리어는 데이터 의존성 배리어나 컨트롤 의존성, ACQUIRE 배리어, RELEASE 배리어, 읽기 배리어, 또는 범용 배리어와 짹을 맞춥니다. 비슷하게 읽기 배리어나 컨트롤 의존성, 또는 데이터 의존성 배리어는 쓰기 배리어나 ACQUIRE 배리어, RELEASE 배리어, 또는 범용 배리어와 짹을 맞추는데, 다음과 같습니다:

CPU 1	CPU 2
=====	=====
WRITE_ONCE(a, 1);	
<쓰기 배리어>	
WRITE_ONCE(b, 2);	x = READ_ONCE(b);

```
<읽기 배리어>
y = READ_ONCE(a);
```

또는:

CPU 1	CPU 2
=====	=====
a = 1;	
<쓰기 배리어>	
WRITE_ONCE(b, &a);	x = READ_ONCE(b);
	<데이터 의존성 배리어>
	y = *x;

또는:

CPU 1	CPU 2
=====	=====
r1 = READ_ONCE(y);	
<범용 배리어>	
WRITE_ONCE(x, 1);	if (r2 = READ_ONCE(x)) {
	<목시적 컨트롤 의존성>
	WRITE_ONCE(y, 1);
}	

```
assert(r1 == 0 || r2 == 0);
```

기본적으로, 여기서의 읽기 배리어는 "더 완화된" 타입일 순 있어도 항상 존재해야 합니다.

[!] 쓰기 배리어 앞의 스토어 오퍼레이션은 일반적으로 읽기 배리어나 데이터 의존성 배리어 뒤의 로드 오퍼레이션과 매치될 것이고, 반대도 마찬가지입니다:

CPU 1	CPU 2
=====	=====
WRITE_ONCE(a, 1);	}---->{ v = READ_ONCE(c);
WRITE_ONCE(b, 2);	} \ / { w = READ_ONCE(d);
<쓰기 배리어>	\ <읽기 배리어>
WRITE_ONCE(c, 3);	/ \ { x = READ_ONCE(a);
WRITE_ONCE(d, 4);	}---->{ y = READ_ONCE(b);

메모리 배리어 시퀀스의 예

첫째, 쓰기 배리어는 스토어 오퍼레이션들의 부분적 순서 세우기로 동작합니다.  
아래의 이벤트 시퀀스를 보세요:

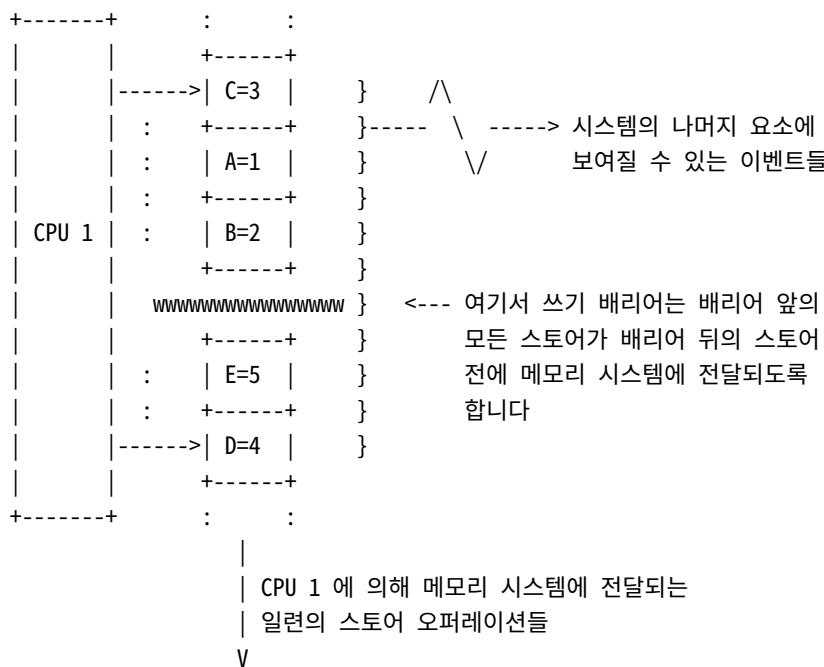
CPU 1	CPU 2
=====	=====
STORE A = 1	
STORE B = 2	
STORE C = 3	

### **<쓰기 배리어>**

STORE D = 4

STORE E = 5

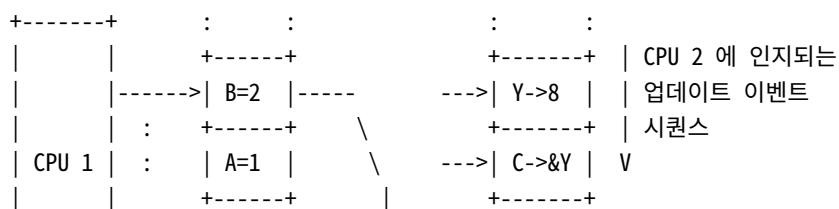
이 이벤트 시퀀스는 메모리 일관성 시스템에 원소끼리의 순서가 존재하지 않는 집합 { STORE A, STORE B, STORE C } 가 역시 원소끼리의 순서가 존재하지 않는 집합 { STORE D, STORE E } 보다 먼저 일어난 것으로 시스템의 나머지 요소들에 보이도록 전달됩니다:

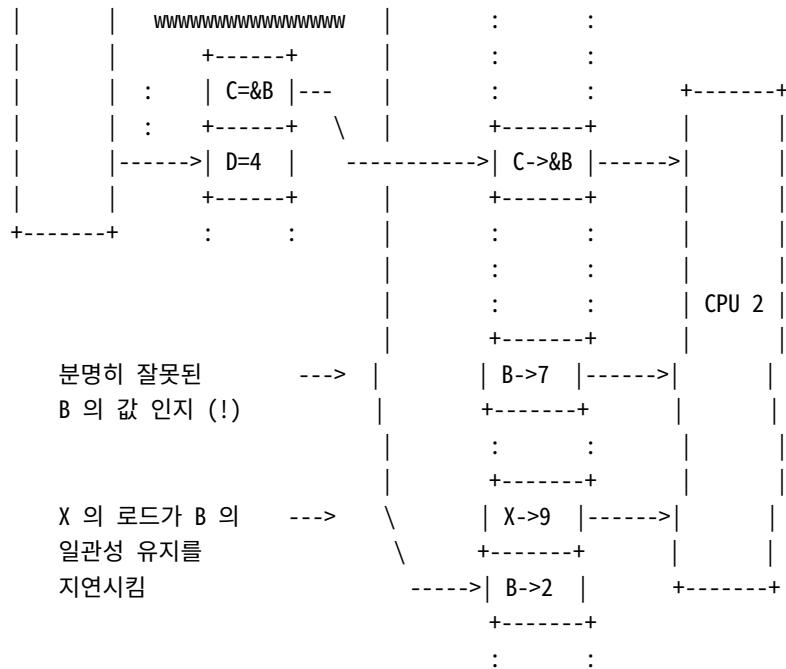


둘째, 데이터의 존성 배리어는 데이터의 존적 로드 오퍼레이션들의 부분적 순서 세우기로 동작합니다. 다음 일련의 이벤트들을 보세요:

CPU 1	CPU 2
=====	=====
{ B = 7; X = 9; Y = 8; C = &Y }	
STORE A = 1	
STORE B = 2	
<쓰기 배리어>	
STORE C = &B	LOAD X
STORE D = 4	LOAD C (gets &B)
	LOAD *C (reads B)

여기에 별다른 개입이 없다면, CPU 1 의 쓰기 배리어에도 불구하고 CPU 2 는 CPU 1 의 이벤트들을 완전히 무작위적 순서로 인지하게 됩니다:



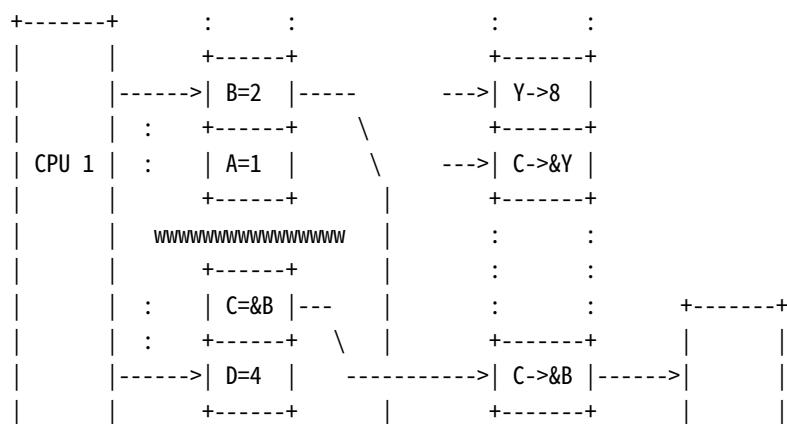


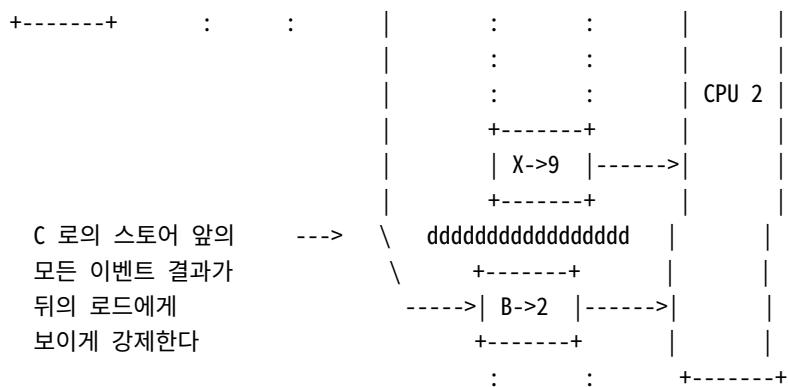
앞의 예에서, CPU 2 는 (B 의 값이 될) \*C 의 값 읽기가 C 의 LOAD 뒤에 이어짐에도 B 가 7 이라는 결과를 얻습니다.

하지만, 만약 데이터 의존성 배리어가 C 의 로드와 \*C (즉, B) 의 로드 사이에 있었다면:

CPU 1 ====== { B = 7; X = 9; Y = 8; C = &Y } STORE A = 1 STORE B = 2 <쓰기 배리어> STORE C = &B STORE D = 4	CPU 2 ====== LOAD X LOAD C (gets &B) <데이터 의존성 배리어> LOAD *C (reads B)
---	---

다음과 같이 됩니다:





셋째, 읽기 배리어는 로드 오퍼레이션들에의 부분적 순서 세우기로 동작합니다.

아래의 일련의 이벤트를 봅시다:

CPU 1                            CPU 2

=====                        =====

{ A = 0, B = 9 }

STORE A=1

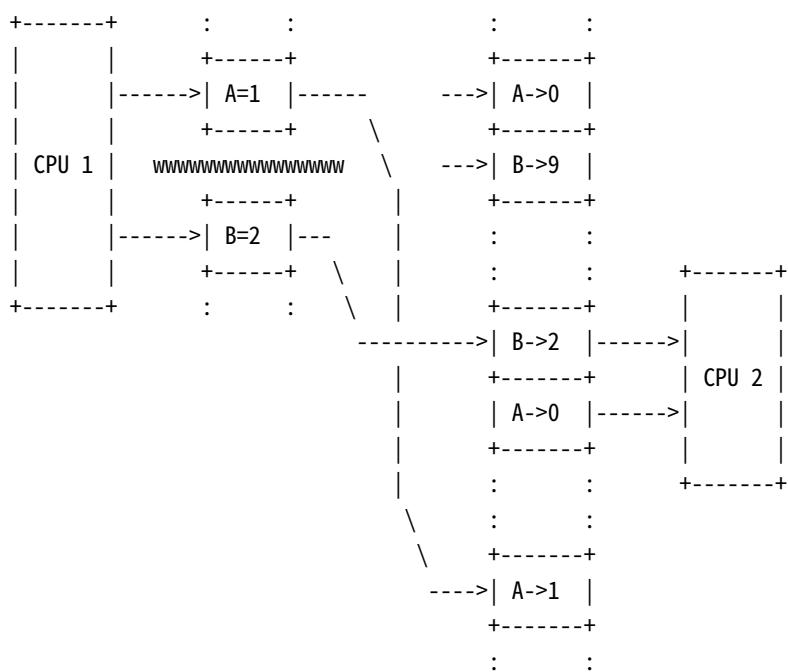
<쓰기 배리어>

STORE B=2

LOAD B

LOAD A

CPU 1 은 쓰기 배리어를 쳤지만, 별다른 개입이 없다면 CPU 2 는 CPU 1 에서 행해진 이벤트의 결과를 무작위적 순서로 인지하게 됩니다.

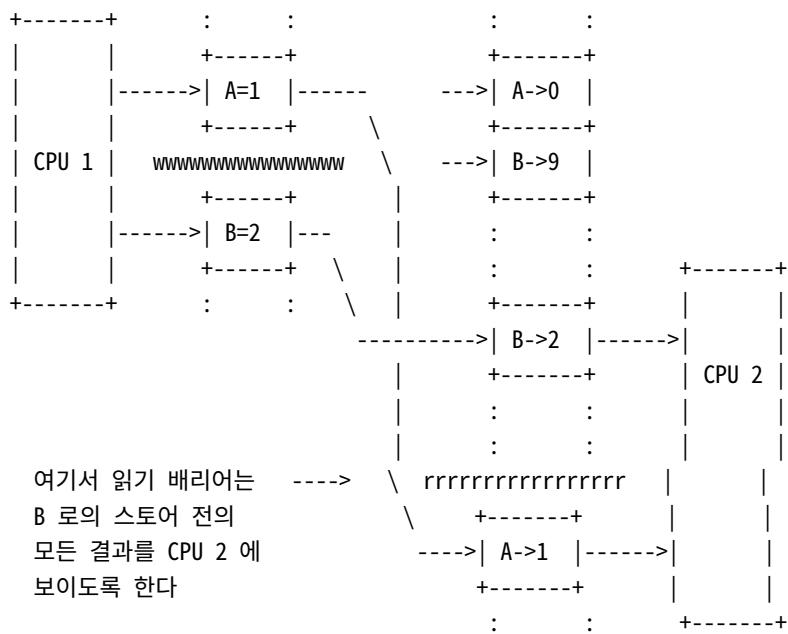


하지만, 만약 읽기 배리어가 B 의 로드와 A 의 로드 사이에 존재한다면:

CPU 1 CPU 2

```
=====
{ A = 0, B = 9 }
STORE A=1
<쓰기 배리어>
STORE B=2
LOAD B
<읽기 배리어>
LOAD A
```

CPU 1에 의해 만들어진 부분적 순서가 CPU 2에도 그대로 인지됩니다:

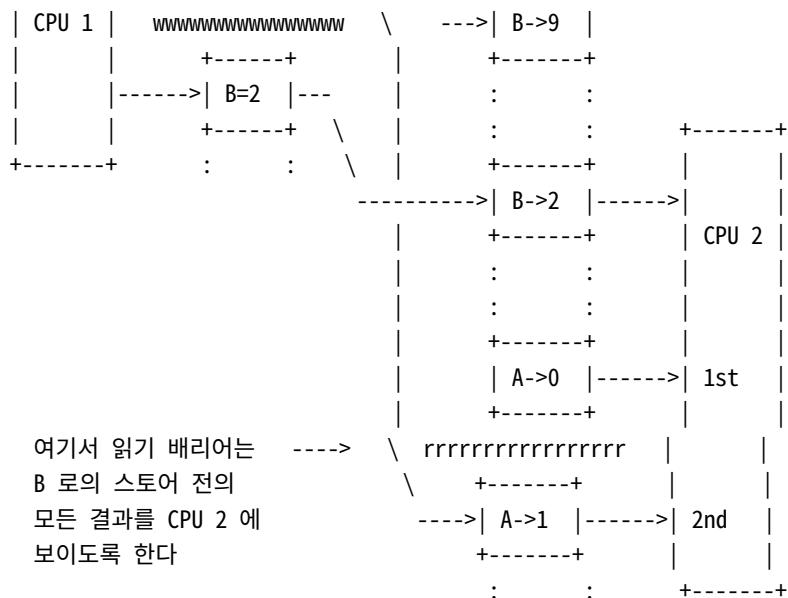


더 완벽한 설명을 위해, A의 로드가 읽기 배리어 앞과 뒤에 있으면 어떻게 될지 생각해 봅시다:

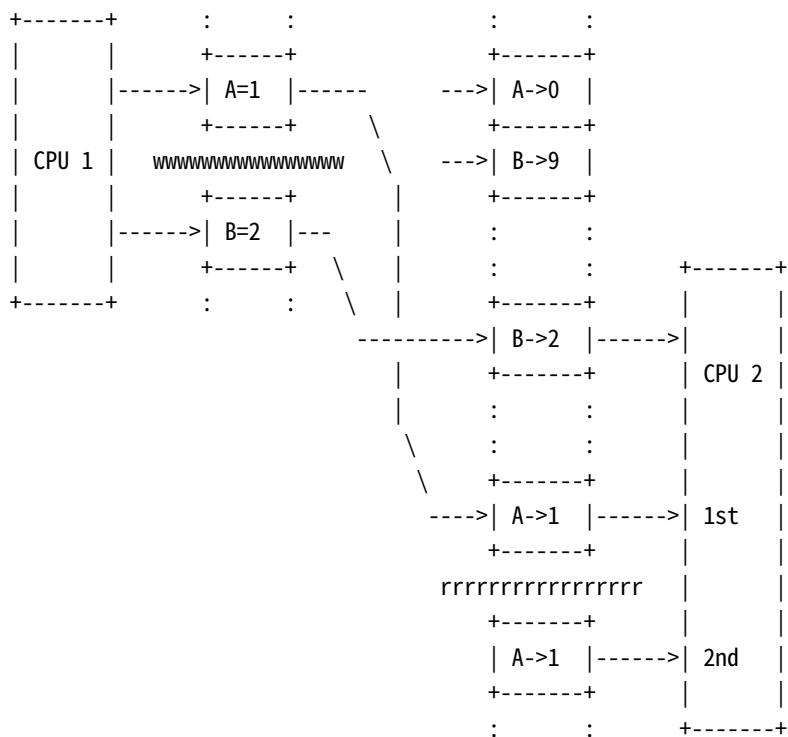
CPU 1	CPU 2
=====	
{ A = 0, B = 9 }	
STORE A=1	
<쓰기 배리어>	
STORE B=2	
LOAD B	
LOAD A [first load of A]	
<읽기 배리어>	
LOAD A [second load of A]	

A의 로드 두개가 모두 B의 로드 뒤에 있지만, 서로 다른 값을 얻어올 수 있습니다:

```
=====
{ A = 0, B = 9 }
STORE A=1
<쓰기 배리어>
STORE B=2
LOAD B
<읽기 배리어>
LOAD A
```



하지만 CPU 1에서의 A 업데이트는 읽기 배리어가 완료되기 전에도 보일 수도 있긴 합니다:



여기서 보장되는 건, 만약 B 의 로드가  $B == 2$  라는 결과를 봤다면, A 에의 두번째 로드는 항상  $A == 1$  을 보게 될 것이라는 겁니다. A 에의 첫번째 로드에는 그런 보장이 없습니다;  $A == 0$  이거나  $A == 1$  이거나 둘 중 하나의 결과를 보게 될겁니다.

읽기 메모리 배리어 VS 로드 예측

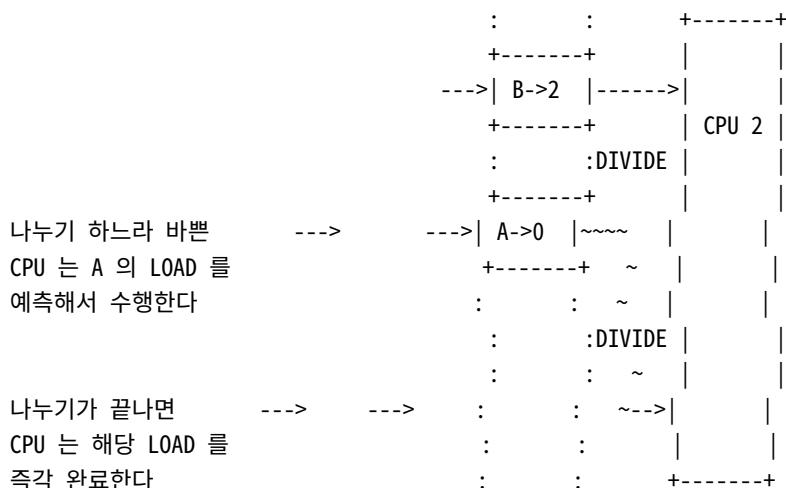
많은 CPU들이 로드를 예측적으로 (speculatively) 합니다: 어떤 데이터를 메모리에서 로드해야 하게 될지 예측을 했다면, 해당 데이터를 로드하는 인스트럭션을 실제로는 아직 만나지 않았더라도 다른 로드 작업이 없어 버스 (bus) 가 아무 일도 하고 있지 않다면, 그 데이터를 로드합니다. 이후에 실제 로드 인스트럭션이 실행되면 CPU 가 이미 그 값을 가지고 있기 때문에 그 로드 인스트럭션은 즉시 완료됩니다.

해당 CPU 는 실제로는 그 값이 필요치 않았다는 사실이 나중에 드러날 수도 있는데 - 해당 로드 인스트럭션이 브랜치로 우회되거나 했을 수 있겠죠 - , 그렇게 되면 앞서 읽어둔 값을 버리거나 나중의 사용을 위해 캐시에 넣어둘 수 있습니다.

다음을 생각해 봅시다:

CPU 1	CPU 2
=====	=====
LOAD B	
DIVIDE	} 나누기 명령은 일반적으로
DIVIDE	} 긴 시간을 필요로 합니다
LOAD A	

는 이렇게 될 수 있습니다:



읽기 배리어나 데이터 의존성 배리어를 두번째 로드 직전에 놓는다면:

CPU 1	CPU 2
=====	=====
LOAD B	
DIVIDE	
DIVIDE	
<읽기 배리어>	
LOAD A	

예측으로 얻어진 값은 사용된 배리어의 타입에 따라서 해당 값이 옳은지 검토되게 됩니다. 만약 해당 메모리 영역에 변화가 없었다면, 예측으로 얻어두었던 값이 사용됩니다:

```

    :      :
    +-----+
    +-----+ | |
    --->| B->2 | ----->| |
    +-----+ | |
    :      :DIVIDE | |
    +-----+ | |
    :      :DIVIDE | |
    +-----+ | |
    :      : ~ | |
    :      : ~ | |
    :      : ~ | |
    rrrrrrrrrrrrrrr~ | |
    :      : ~ | |
    :      : ~-->| |
    :      : | |
    :      : | |

```

나누기 하느라 바쁜  
CPU 는 A 의 LOAD 를  
예측한다

하지만 다른 CPU에서 업데이트나 무효화가 있었다면, 그 예측은 무효화되고 그 값은 다시 읽혀집니다:

The diagram illustrates the execution of a divide operation on CPU 2. It shows the state of registers A and B across four stages:

- Initial State:** Register A contains 0 and register B contains 2.
- After Load:** Register A is updated to 0 and register B is updated to 2. This stage is labeled "CPU 는 A 의 LOAD 를 예측한다".
- After Division:** Register A is updated to 1 and register B is updated to 0. This stage is labeled "예측성 동작은 무효화 되고 업데이트된 값이 다시 읽혀진다".
- Final Result:** Register A contains 1.

MULTICOPY 원자성

**Multipcopy** 원자성은 실제의 컴퓨터 시스템에서 항상 제공되지는 않는, 순서 맞추기에 대한 상당히 직관적인 개념으로, 특정 스토어가 모든 CPU 들에게 동시에 보여지게 됨을, 달리 말하자면 모든 CPU 들이 모든 스토어들이 보여지는 순서를 동의하게 되는 것입니다. 하지만, 완전한 **multipcopy** 원자성의 사용은 가치있는 하드웨어 최적화들을 무능하게 만들아버릴 수 있어서, 보다 완화된 형태의 ``다른 **multipcopy** 원자성'' 라는 이름의, 특정 스토어가 모든 -다른- CPU 들에게는 동시에 보여지게 하는 보장을 대신 제공합니다. 이 문서의 뒷부분들은 이 완화된 형태에 대해 논하게 됩니다만, 단순히 ``**multipcopy** 원자성'' 이라고 부르겠습니다.

다음의 예가 multicopy 원자성을 보입니다:

CPU 1	CPU 2	CPU 3
{ X = 0, Y = 0 }		
STORE X=1	r1=LOAD X (reads 1) <범용 배리어>	LOAD Y (reads 1) <읽기 배리어>
	STORE Y=r1	LOAD X

CPU 2 의 Y 로의 스토어에 사용되는 X 로드의 결과가 1 이었고 CPU 3 의 Y 로드가 1을 리턴했다고 해봅시다. 이는 CPU 1 의 X 로의 스토어가 CPU 2 의 X 로부터의 로드를 앞서고 CPU 2 의 Y 로의 스토어가 CPU 3 의 Y 로부터의 로드를 앞섬을 의미합니다. 또한, 여기서의 메모리 배리어들은 CPU 2 가 자신의 로드를 자신의 스토어 전에 수행하고, CPU 3 가 Y 로부터의 로드를 X 로부터의 로드 전에 수행함을 보장합니다. 그럼 "CPU 3 의 X 로부터의 로드는 0 을 리턴할 수 있을까요?"

CPU 3 의 X 로드가 CPU 2 의 로드보다 뒤에 이루어졌으므로, CPU 3 의 X 로부터의 로드는 1 을 리턴한다고 예상하는게 당연합니다. 이런 예상은 multicopy 원자성으로부터 나옵니다: CPU B 에서 수행된 로드가 CPU A 의 같은 변수로부터의 로드를 뒤따른다면 (그리고 CPU A 가 자신이 읽은 값으로 먼저 해당 변수에 스토어 하지 않았다면) multicopy 원자성을 제공하는 시스템에서는, CPU B 의 로드가 CPU A 의 로드와 같은 값 또는 그 나중 값을 리턴해야만 합니다. 하지만, 리눅스 커널은 시스템들이 multicopy 원자성을 제공할 것을 요구하지 않습니다.

앞의 범용 메모리 배리어의 사용은 모든 multicopy 원자성의 부족을 보상해줍니다. 앞의 예에서, CPU 2 의 X 로부터의 로드가 1 을 리턴했고 CPU 3 의 Y 로부터의 로드가 1 을 리턴했다면, CPU 3 의 X 로부터의 로드는 1 을 리턴해야만 합니다.

하지만, 의존성, 읽기 배리어, 쓰기 배리어는 항상 non-multicopy 원자성을 보상해 주지는 않습니다. 예를 들어, CPU 2 의 범용 배리어가 앞의 예에서 사라져서 아래처럼 데이터 의존성만 남게 되었다고 해봅시다:

CPU 1	CPU 2	CPU 3
{ X = 0, Y = 0 }		
STORE X=1	r1=LOAD X (reads 1) <데이터 의존성>	LOAD Y (reads 1) <읽기 배리어>
	STORE Y=r1	LOAD X (reads 0)

이 변화는 non-multicopy 원자성이 만연하게 합니다: 이 예에서, CPU 2 의 X 로부터의 로드가 1 을 리턴하고, CPU 3 의 Y 로부터의 로드가 1 을 리턴하는데, CPU 3 의 X 로부터의 로드가 0 을 리턴하는게 완전히 합법적입니다.

핵심은, CPU 2 의 데이터 의존성이 자신의 로드와 스토어를 순서짓지만, CPU 1 의 스토어에 대한 순서는 보장하지 않는다는 것입니다. 따라서, 이 예제가 CPU 1 과 CPU 2 가 스토어 버퍼나 한 수준의 캐시를 공유하는, multicopy 원자성을 제공하지 않는 시스템에서 수행된다면 CPU 2 는 CPU 1 의 쓰기에 이를 접근할 수도 있습니다. 따라서, 모든 CPU 들이 여러 접근들의 조합된 순서에 대해서 동의하게 하기 위해서는 범용 배리어가 필요합니다.

범용 배리어는 non-multicopy 원자성만 보상할 수 있는게 아니라, -모든- CPU 들이

-모든- 오퍼레이션들의 순서를 동일하게 인식하게 하는 추가적인 순서 보장을 만들어냅니다. 반대로, release-acquire 짹의 연결은 이런 추가적인 순서는 제공하지 않는데, 해당 연결에 들어있는 CPU 들만이 메모리 접근의 조합된 순서에 대해 동의할 것으로 보장됨을 의미합니다. 예를 들어, 존경스런 Herman Hollerith 의 코드를 C 코드로 변환하면:

```
int u, v, x, y, z;

void cpu0(void)
{
    r0 = smp_load_acquire(&x);
    WRITE_ONCE(u, 1);
    smp_store_release(&y, 1);
}

void cpu1(void)
{
    r1 = smp_load_acquire(&y);
    r4 = READ_ONCE(v);
    r5 = READ_ONCE(u);
    smp_store_release(&z, 1);
}

void cpu2(void)
{
    r2 = smp_load_acquire(&z);
    smp_store_release(&x, 1);
}

void cpu3(void)
{
    WRITE_ONCE(v, 1);
    smp_mb();
    r3 = READ_ONCE(u);
}
```

cpu0(), cpu1(), 그리고 cpu2() 는 smp\_store\_release()/smp\_load\_acquire() 쌍의 연결에 참여되어 있으므로, 다음과 같은 결과는 나오지 않을 겁니다:

```
r0 == 1 && r1 == 1 && r2 == 1
```

더 나아가서, cpu0() 와 cpu1() 사이의 release-acquire 관계로 인해, cpu1() 은 cpu0() 의 쓰기를 봐야만 하므로, 다음과 같은 결과도 없을 겁니다:

```
r1 == 1 && r5 == 0
```

하지만, release-acquire 에 의해 제공되는 순서는 해당 연결에 동참한 CPU 들에만 적용되므로 cpu3() 에, 적어도 스토어들 외에는 적용되지 않습니다. 따라서, 다음과 같은 결과가 가능합니다:

```
r0 == 0 && r1 == 1 && r2 == 1 && r3 == 0 && r4 == 0
```

비슷하게, 다음과 같은 결과도 가능합니다:

```
r0 == 0 && r1 == 1 && r2 == 1 && r3 == 0 && r4 == 0 && r5 == 1
```

cpu0(), cpu1(), 그리고 cpu2() 는 그들의 읽기와 쓰기를 순서대로 보게 되지만, *release-acquire* 체인에 관여되지 않은 CPU 들은 그 순서에 이견을 가질 수 있습니다. 이런 이견은 *smp\_load\_acquire()* 와 *smp\_store\_release()* 의 구현에 사용되는 완화된 메모리 배리어 인스트럭션들은 항상 배리어 앞의 스토어들을 뒤의 로드들에 앞세울 필요는 없다는 사실에서 기인합니다. 이 말은 cpu3() 는 cpu0() 의 u 로의 스토어를 cpu1() 의 v 로부터의 로드 뒤에 일어난 것으로 볼 수 있다는 뜻입니다, cpu0() 와 cpu1() 은 이 두 오퍼레이션이 의도된 순서대로 일어났음에 모두 동의하는데도 말입니다.

하지만, *smp\_load\_acquire()* 는 마술이 아님을 명심하시기 바랍니다. 구체적으로, 이 함수는 단순히 순서 규칙을 지키며 인자로부터의 읽기를 수행합니다. 이것은 어떤 특정한 값이 읽힐 것인지는 보장하지 -않습니다-. 따라서, 다음과 같은 결과도 가능합니다:

```
r0 == 0 && r1 == 0 && r2 == 0 && r5 == 0
```

이런 결과는 어떤 것도 재배치 되지 않는, 순차적 일관성을 가진 가상의 시스템에서도 일어날 수 있음을 기억해 두시기 바랍니다.

다시 말하지만, 당신의 코드가 모든 오퍼레이션들의 완전한 순서를 필요로 한다면, 범용 배리어를 사용하십시오.

---

### ===== 명시적 커널 배리어 =====

리눅스 커널은 서로 다른 단계에서 동작하는 다양한 배리어들을 가지고 있습니다:

(\*) 컴파일러 배리어.

(\*) CPU 메모리 배리어.

---

### 컴파일러 배리어

---

리눅스 커널은 컴파일러가 메모리 액세스를 재배치 하는 것을 막아주는 명시적인 컴파일러 배리어를 가지고 있습니다:

```
barrier();
```

이건 범용 배리어입니다 -- *barrier()* 의 읽기-읽기 나 쓰기-쓰기 변종은 없습니다. 하지만, *READ\_ONCE()* 와 *WRITE\_ONCE()* 는 특정 액세스들에 대해서만 동작하는 *barrier()* 의 완화된 형태로 볼 수 있습니다.

*barrier()* 함수는 다음과 같은 효과를 갖습니다:

(\*) 컴파일러가 `barrier()` 뒤의 액세스들이 `barrier()` 앞의 액세스보다 앞으로 재배치되지 못하게 합니다. 예를 들어, 인터럽트 핸들러 코드와 인터럽트 당한 코드 사이의 통신을 신중히 하기 위해 사용될 수 있습니다.

(\*) 루프에서, 컴파일러가 루프 조건에 사용된 변수를 매 이터레이션마다 메모리에서 로드하지 않아도 되도록 최적화 하는 걸 방지합니다.

`READ_ONCE()` 와 `WRITE_ONCE()` 함수는 싱글 쓰레드 코드에서는 문제 없지만 동시성이 있는 코드에서는 문제가 될 수 있는 모든 최적화를 막습니다. 이런 류의 최적화에 대한 예를 몇 가지 들어보면 다음과 같습니다:

(\*) 컴파일러는 같은 변수에 대한 로드와 스토어를 재배치 할 수 있고, 어떤 경우에는 CPU가 같은 변수로부터의 로드들을 재배치할 수도 있습니다. 이는 다음의 코드가:

```
a[0] = x;
a[1] = x;
```

x 의 예전 값이 a[1] 에, 새 값이 a[0] 에 있게 할 수 있다는 뜻입니다.

컴파일러와 CPU가 이런 일을 못하게 하려면 다음과 같이 해야 합니다:

```
a[0] = READ_ONCE(x);
a[1] = READ_ONCE(x);
```

즉, `READ_ONCE()` 와 `WRITE_ONCE()` 는 여러 CPU 에서 하나의 변수에 가해지는 액세스들에 캐시 일관성을 제공합니다.

(\*) 컴파일러는 같은 변수에 대한 연속적인 로드들을 병합할 수 있습니다. 그런 병합 작업으로 컴파일러는 다음의 코드를:

```
while (tmp = a)
    do_something_with(tmp);
```

다음과 같이, 싱글 쓰레드 코드에서는 말이 되지만 개발자의 의도와 전혀 맞지 않는 방향으로 "최적화" 할 수 있습니다:

```
if (tmp = a)
    for (;;)
        do_something_with(tmp);
```

컴파일러가 이런 짓을 하지 못하게 하려면 `READ_ONCE()` 를 사용하세요:

```
while (tmp = READ_ONCE(a))
    do_something_with(tmp);
```

(\*) 예컨대 레지스터 사용량이 많아 컴파일러가 모든 데이터를 레지스터에 담을 수 없는 경우, 컴파일러는 변수를 다시 로드할 수 있습니다. 따라서 컴파일러는 앞의 예에서 변수 '`tmp`' 사용을 최적화로 없애버릴 수 있습니다:

```
while (tmp = a)
    do_something_with(tmp);
```

이 코드는 다음과 같이 싱글 쓰레드에서는 완벽하지만 동시성이 존재하는 경우엔 치명적인 코드로 바뀔 수 있습니다:

```
while (a)
    do_something_with(a);
```

예를 들어, 최적화된 이 코드는 변수 a 가 다른 CPU 에 의해 "while" 문과 do\_something\_with() 호출 사이에 바뀌어 do\_something\_with() 에 0을 넘길 수도 있습니다.

이번에도, 컴파일러가 그런 짓을 하는걸 막기 위해 READ\_ONCE() 를 사용하세요:

```
while (tmp = READ_ONCE(a))
    do_something_with(tmp);
```

레지스터가 부족한 상황을 겪는 경우, 컴파일러는 tmp 를 스택에 저장해둘 수도 있습니다. 컴파일러가 변수를 다시 읽어들이는건 이렇게 저장해두고 후에 다시 읽어들이는데 드는 오버헤드 때문입니다. 그렇게 하는게 싱글 쓰레드 코드에서는 안전하므로, 안전하지 않은 경우에는 컴파일러에게 직접 알려줘야 합니다.

(\*) 컴파일러는 그 값이 무엇일지 알고 있다면 로드를 아예 안할 수도 있습니다.

예를 들어, 다음의 코드는 변수 'a' 의 값이 항상 0임을 증명할 수 있다면:

```
while (tmp = a)
    do_something_with(tmp);
```

이렇게 최적화 되어버릴 수 있습니다:

```
do { } while (0);
```

이 변환은 싱글 쓰레드 코드에서는 도움이 되는데 로드와 브랜치를 제거했기 때문입니다. 문제는 컴파일러가 'a' 의 값을 업데이트 하는건 현재의 CPU 하나 뿐이라는 가정 위에서 증명을 했다는데 있습니다. 만약 변수 'a' 가 공유되어 있다면, 컴파일러의 증명은 틀린 것이 될겁니다. 컴파일러는 그 자신이 생각하는 것만큼 많은 것을 알고 있지 못함을 컴파일러에게 알리기 위해 READ\_ONCE() 를 사용하세요:

```
while (tmp = READ_ONCE(a))
    do_something_with(tmp);
```

하지만 컴파일러는 READ\_ONCE() 뒤에 나오는 값에 대해서도 눈길을 두고 있음을 기억하세요. 예를 들어, 다음의 코드에서 MAX 는 전처리기 매크로로, 1의 값을 갖는다고 해봅시다:

```
while ((tmp = READ_ONCE(a)) % MAX)
    do_something_with(tmp);
```

이렇게 되면 컴파일러는 MAX 를 가지고 수행되는 "%" 오퍼레이터의 결과가 항상 0이라는 것을 알게 되고, 컴파일러가 코드를 실질적으로는 존재하지 않는 것처럼 최적화 하는 것이 허용되어 버립니다. ('a' 변수의 로드는 여전히 행해질 겁니다.)

- (\*) 비슷하게, 컴파일러는 변수가 저장하려 하는 값을 이미 가지고 있다는 것을 알면 스토어 자체를 제거할 수 있습니다. 이번에도, 컴파일러는 현재의 CPU 만이 그 변수에 값을 쓰는 오로지 하나의 존재라고 생각하여 공유된 변수에 대해서는 잘못된 일을 하게 됩니다. 예를 들어, 다음과 같은 경우가 있을 수 있습니다:

```
a = 0;
... 변수 a 에 스토어를 하지 않는 코드 ...
a = 0;
```

컴파일러는 변수 'a' 의 값은 이미 0이라는 것을 알고, 따라서 두번째 스토어를 삭제할 겁니다. 만약 다른 CPU 가 그 사이 변수 'a' 에 다른 값을 썼다면 황당한 결과가 나올 겁니다.

컴파일러가 그런 잘못된 추측을 하지 않도록 `WRITE_ONCE()` 를 사용하세요:

```
WRITE_ONCE(a, 0);
... 변수 a 에 스토어를 하지 않는 코드 ...
WRITE_ONCE(a, 0);
```

- (\*) 컴파일러는 하지 말라고 하지 않으면 메모리 액세스들을 재배치 할 수 있습니다. 예를 들어, 다음의 프로세스 레벨 코드와 인터럽트 핸들러 사이의 상호작용을 생각해 봅시다:

```
void process_level(void)
{
    msg = get_message();
    flag = true;
}

void interrupt_handler(void)
{
    if (flag)
        process_message(msg);
}
```

이 코드에는 컴파일러가 `process_level()` 을 다음과 같이 변환하는 것을 막을 수단이 없고, 이런 변환은 싱글쓰레드에서라면 실제로 훌륭한 선택일 수 있습니다:

```
void process_level(void)
{
    flag = true;
    msg = get_message();
}
```

이 두개의 문장 사이에 인터럽트가 발생한다면, `interrupt_handler()` 는 의미를 알 수 없는 메세지를 받을 수도 있습니다. 이걸 막기 위해 다음과 같이 `WRITE_ONCE()` 를 사용하세요:

```
void process_level(void)
```

```
{  
    WRITE_ONCE(msg, get_message());  
    WRITE_ONCE(flag, true);  
}  
  
void interrupt_handler(void)  
{  
    if (READ_ONCE(flag))  
        process_message(READ_ONCE(msg));  
}
```

`interrupt_handler()` 안에서도 중첩된 인터럽트나 NMI 와 같이 인터럽트 핸들러 역시 'flag' 와 'msg' 에 접근하는 또다른 무언가에 인터럽트 될 수 있다면 `READ_ONCE()` 와 `WRITE_ONCE()` 를 사용해야 함을 기억해 두세요. 만약 그런 가능성이 없다면, `interrupt_handler()` 안에서는 문서화 목적이 아니라면 `READ_ONCE()` 와 `WRITE_ONCE()` 는 필요치 않습니다. (근래의 리눅스 커널에서 중첩된 인터럽트는 보통 잘 일어나지 않음도 기억해 두세요, 실제로, 어떤 인터럽트 핸들러가 인터럽트가 활성화된 채로 리턴하면 `WARN_ONCE()` 가 실행됩니다.)

컴파일러는 `READ_ONCE()` 와 `WRITE_ONCE()` 뒤의 `READ_ONCE()` 나 `WRITE_ONCE()`, `barrier()`, 또는 비슷한 것들을 담고 있지 않은 코드를 움직일 수 있을 것으로 가정되어야 합니다.

이 효과는 `barrier()` 를 통해서도 만들 수 있지만, `READ_ONCE()` 와 `WRITE_ONCE()` 가 좀 더 안목 높은 선택입니다: `READ_ONCE()` 와 `WRITE_ONCE()` 는 컴파일러에 주어진 메모리 영역에 대해서만 최적화 가능성을 포기하도록 하지만, `barrier()` 는 컴파일러가 지금까지 기계의 레지스터에 캐시해 놓은 모든 메모리 영역의 값을 버려야 하게 하기 때문입니다. 물론, 컴파일러는 `READ_ONCE()` 와 `WRITE_ONCE()` 가 일어난 순서도 지켜줍니다, CPU 는 당연히 그 순서를 지킬 의무가 없지만요.

(\*) 컴파일러는 다음의 예에서와 같이 변수에의 스토어를 날조해낼 수도 있습니다:

```
if (a)  
    b = a;  
else  
    b = 42;
```

컴파일러는 아래와 같은 최적화로 브랜치를 줄일 겁니다:

```
b = 42;  
if (a)  
    b = a;
```

싱글 쓰레드 코드에서 이 최적화는 안전할 뿐 아니라 브랜치 갯수를 줄여줍니다. 하지만 안타깝게도, 동시성이 있는 코드에서는 이 최적화는 다른 CPU 가 'b' 를 로드할 때, -- 'a' 가 0이 아닌데도 -- 가짜인 값, 42를 보게 되는 경우를 가능하게 합니다. 이걸 방지하기 위해 `WRITE_ONCE()` 를 사용하세요:

```
if (a)
```

```

    WRITE_ONCE(b, a);
else
    WRITE_ONCE(b, 42);

```

컴파일러는 로드를 만들어낼 수도 있습니다. 일반적으로는 문제를 일으키지 않지만, 캐시 라인 바운싱을 일으켜 성능과 확장성을 떨어뜨릴 수 있습니다. 날조된 로드를 막기 위해선 READ\_ONCE() 를 사용하세요.

- (\*) 정렬된 메모리 주소에 위치한, 한번의 메모리 참조 인스트럭션으로 액세스 가능한 크기의 데이터는 하나의 큰 액세스가 여러개의 작은 액세스들로 대체되는 "로드 티어링(load tearing)" 과 "스토어 티어링(store tearing)" 을 방지합니다. 예를 들어, 주어진 아키텍쳐가 7-bit immediate field 를 갖는 16-bit 스토어 인스트럭션을 제공한다면, 컴파일러는 다음의 32-bit 스토어를 구현하는데에 두개의 16-bit store-immediate 명령을 사용하려 할겁니다:

```
p = 0x00010002;
```

스토어 할 상수를 만들고 그 값을 스토어 하기 위해 두개가 넘는 인스트럭션을 사용하게 되는, 이런 종류의 최적화를 GCC 는 실제로 함을 부디 알아 두십시오. 이 최적화는 싱글 쓰레드 코드에서는 성공적인 최적화 입니다. 실제로, 근래에 발생한 (그리고 고쳐진) 버그는 GCC 가 volatile 스토어에 비정상적으로 이 최적화를 사용하게 했습니다. 그런 버그가 없다면, 다음의 예에서 WRITE\_ONCE() 의 사용은 스토어 티어링을 방지합니다:

```
WRITE_ONCE(p, 0x00010002);
```

Packed 구조체의 사용 역시 다음의 예처럼 로드 / 스토어 티어링을 유발할 수 있습니다:

```

struct __attribute__((__packed__)) foo {
    short a;
    int b;
    short c;
};

struct foo foo1, foo2;
...

foo2.a = foo1.a;
foo2.b = foo1.b;
foo2.c = foo1.c;

```

READ\_ONCE() 나 WRITE\_ONCE() 도 없고 volatile 마킹도 없기 때문에, 컴파일러는 이 세개의 대입문을 두개의 32-bit 로드와 두개의 32-bit 스토어로 변환할 수 있습니다. 이는 'foo1.b' 의 값의 로드 티어링과 'foo2.b' 의 스토어 티어링을 초래할 겁니다. 이 예에서도 READ\_ONCE() 와 WRITE\_ONCE() 가 티어링을 막을 수 있습니다:

```

foo2.a = foo1.a;
WRITE_ONCE(foo2.b, READ_ONCE(foo1.b));
foo2.c = foo1.c;

```

그렇지만, volatile 로 마크된 변수에 대해서는 READ\_ONCE() 와 WRITE\_ONCE() 가

필요치 않습니다. 예를 들어, 'jiffies' 는 volatile 로 마크되어 있기 때문에, READ\_ONCE(jiffies) 라고 할 필요가 없습니다. READ\_ONCE() 와 WRITE\_ONCE() 가 실은 volatile 캐스팅으로 구현되어 있어서 인자가 이미 volatile 로 마크되어 있다면 또다른 효과를 내지는 않기 때문입니다.

이 컴파일러 배리어들은 CPU 에는 직접적 효과를 전혀 만들지 않기 때문에, 결국은 재배치가 일어날 수도 있음을 부디 기억해 두십시오.

### CPU 메모리 배리어

---

리눅스 커널은 다음의 여덟개 기본 CPU 메모리 배리어를 가지고 있습니다:

TYPE	MANDATORY	SMP CONDITIONAL
범용	mb()	smp_mb()
쓰기	wmb()	smp_wmb()
읽기	rmb()	smp_rmb()
데이터 의존성		READ_ONCE()

데이터 의존성 배리어를 제외한 모든 메모리 배리어는 컴파일러 배리어를 포함합니다. 데이터 의존성은 컴파일러에의 추가적인 순서 보장을 포함하지 않습니다.

방백: 데이터 의존성이 있는 경우, 컴파일러는 해당 로드를 올바른 순서로 일으킬 것으로 (예: `a[b]` 는 a[b] 를 로드 하기 전에 b 의 값을 먼저 로드한다) 기대되지만, C 언어 사양에는 컴파일러가 b 의 값을 추측 (예: 1 과 같음) 해서 b 로드 전에 a 로드를 하는 코드 (예: tmp = a[1]; if (b != 1) tmp = a[b]; ) 를 만들지 않아야 한다는 내용 같은 건 없습니다. 또한 컴파일러는 a[b] 를 로드한 후에 b 를 또다시 로드할 수도 있어서, a[b] 보다 최신 버전의 b 값을 가질 수도 있습니다. 이런 문제들의 해결책에 대한 의견 일치는 아직 없습니다만, 일단 READ\_ONCE() 매크로부터 보기 시작하는게 좋은 시작이 될겁니다.

SMP 메모리 배리어들은 유니프로세서로 컴파일된 시스템에서는 컴파일러 배리어로 바뀌는데, 하나의 CPU 는 스스로 일관성을 유지하고, 겹치는 액세스들 역시 올바른 순서로 행해질 것으로 생각되기 때문입니다. 하지만, 아래의 "Virtual Machine Guests" 서브섹션을 참고하십시오.

[!] SMP 시스템에서 공유메모리로의 접근들을 순서 세워야 할 때, SMP 메모리 배리어는 반드시 사용되어야 함을 기억하세요, 그대신 락을 사용하는 것으로도 충분하긴 하지만 말이죠.

Mandatory 배리어들은 SMP 시스템에서도 UP 시스템에서도 SMP 효과만 통제하기에는 불필요한 오버헤드를 갖기 때문에 SMP 효과만 통제하면 되는 곳에는 사용되지 않아야 합니다. 하지만, 느슨한 순서 규칙의 메모리 I/O 윈도우를 통한 MMIO 의 효과를 통제할 때에는 mandatory 배리어들이 사용될 수 있습니다. 이 배리어들은 컴파일러와 CPU 모두 재배치를 못하도록 함으로써 메모리 오퍼레이션들이 디바이스에 보여지는 순서에도 영향을 주기 때문에, SMP 가 아닌 시스템이라 할지라도 필요할 수 있습니다.

일부 고급 배리어 함수들도 있습니다:

```
(*) smp_store_mb(var, value)
```

이 함수는 특정 변수에 특정 값을 대입하고 범용 메모리 배리어를 칩니다.

UP 컴파일에서는 컴파일러 배리어보다 더한 것을 친다고는 보장되지 않습니다.

```
(*) smp_mb__before_atomic();
(*) smp_mb__after_atomic();
```

이것들은 메모리 배리어를 내포하지 않는 어토믹 RMW 함수를 사용하지만 코드에 메모리 배리어가 필요한 경우를 위한 것들입니다. 메모리 배리어를 내포하지 않는 어토믹 RMW 함수들의 예로는 더하기, 빼기, (실패한) 조건적 오퍼레이션들, \_relaxed 함수들이 있으며, atomic\_read 나 atomic\_set 은 이에 해당되지 않습니다. 메모리 배리어가 필요해지는 흔한 예로는 어토믹 오퍼레이션을 사용해 레퍼런스 카운트를 수정하는 경우를 들 수 있습니다.

이것들은 또한 (set\_bit 과 clear\_bit 같은) 메모리 배리어를 내포하지 않는 어토믹 RMW bitop 함수들을 위해서도 사용될 수 있습니다.

한 예로, 객체 하나를 무효한 것으로 표시하고 그 객체의 레퍼런스 카운트를 감소시키는 다음 코드를 보세요:

```
obj->dead = 1;
smp_mb__before_atomic();
atomic_dec(&obj->ref_count);
```

이 코드는 객체의 업데이트된 death 마크가 레퍼런스 카운터 감소 동작 \*전에\* 보일 것을 보장합니다.

더 많은 정보를 위해선 Documentation/atomic\_{t,bitops}.txt 문서를 참고하세요.

```
(*) dma_wmb();
(*) dma_rmb();
```

이것들은 CPU 와 DMA 가능한 디바이스에서 모두 액세스 가능한 공유 메모리의 읽기, 쓰기 작업들의 순서를 보장하기 위해 consistent memory 에서 사용하기 위한 것들입니다.

예를 들어, 디바이스와 메모리를 공유하며, 디스크립터 상태 값을 사용해 디스크립터가 디바이스에 속해 있는지 아니면 CPU 에 속해 있는지 표시하고, 공지용 초인종(doorbell) 을 사용해 업데이트된 디스크립터가 디바이스에 사용 가능해졌음을 공지하는 디바이스 드라이버를 생각해 봅시다:

```
if (desc->status != DEVICE_OWN) {
    /* 디스크립터를 소유하기 전에는 데이터를 읽지 않음 */
    dma_rmb();
```

```
/* 데이터를 읽고 씀 */
read_data = desc->data;
desc->data = write_data;

/* 상태 업데이트 전 수정사항을 반영 */
dma_wmb();

/* 소유권을 수정 */
desc->status = DEVICE_OWN;

/* 업데이트된 디스크립터의 디바이스에 공지 */
writel(DESC_NOTIFY, doorbell);
}
```

`dma_rmb()` 는 디스크립터로부터 데이터를 읽어오기 전에 디바이스가 소유권을 내려놓았을 것을 보장하고, `dma_wmb()` 는 디바이스가 자신이 소유권을 다시 가져옴을 보기 전에 디스크립터에 데이터가 쓰였을 것을 보장합니다. 참고로, `writel()` 을 사용하면 캐시 일관성이 있는 메모리 (cache coherent memory) 쓰기가 MMIO 영역에의 쓰기 전에 완료되었을 것을 보장하므로 `writel()` 앞에 `wmb()` 를 실행할 필요가 없음을 알아두시기 바랍니다. `writel()` 보다 비용이 저렴한 `writel_relaxed()` 는 이런 보장을 제공하지 않으므로 여기선 사용되지 않아야 합니다.

`writel_relaxed()` 와 같은 완화된 I/O 접근자들에 대한 자세한 내용을 위해서는 "커널 I/O 배리어의 효과" 섹션을, `consistent memory` 에 대한 자세한 내용을 위해선 `Documentation/core-api/dma-api.rst` 문서를 참고하세요.

(\*) `pmem_wmb();`

이것은 `persistent memory` 를 위한 것으로, `persistent` 저장소에 가해진 변경 사항이 플랫폼 연속성 도메인에 도달했을 것을 보장하기 위한 것입니다.

예를 들어, 임시적이지 않은 `pmem` 영역으로의 쓰기 후, 우리는 쓰기가 플랫폼 연속성 도메인에 도달했을 것을 보장하기 위해 `pmem_wmb()` 를 사용합니다. 이는 쓰기가 뒤따르는 `instruction` 들이 유발하는 어떠한 데이터 액세스나 데이터 전송의 시작 전에 `persistent` 저장소를 업데이트 했을 것을 보장합니다. 이는 `wmb()` 에 의해 이뤄지는 순서 규칙을 포함합니다.

`Persistent memory` 에서의 로드를 위해선 현재의 읽기 메모리 배리어로도 읽기 순서를 보장하는데 충분합니다.

---

=====  
암묵적 커널 메모리 배리어  
=====

리눅스 커널의 일부 함수들은 메모리 배리어를 내장하고 있는데, 락(lock)과 스케줄링 관련 함수들이 대부분입니다.

여기선 최소한의 보장을 설명합니다; 특정 아키텍쳐에서는 이 설명보다 더 많은 보장을 제공할 수도 있습니다만 해당 아키텍쳐에 종속적인 코드 외의 부분에서는 그런 보장을 기대해선 안될겁니다.

## 락 ACQUISITION 함수

---

리눅스 커널은 다양한 락 구성체를 가지고 있습니다:

- (\*) 스핀 락
- (\*) R/W 스핀 락
- (\*) 뮤텍스
- (\*) 세마포어
- (\*) R/W 세마포어

각 구성체마다 모든 경우에 "ACQUIRE" 오퍼레이션과 "RELEASE" 오퍼레이션의 변종이 존재합니다. 이 오퍼레이션들은 모두 적절한 배리어를 내포하고 있습니다:

### (1) ACQUIRE 오퍼레이션의 영향:

ACQUIRE 뒤에서 요청된 메모리 오퍼레이션은 ACQUIRE 오퍼레이션이 완료된 뒤에 완료됩니다.

ACQUIRE 앞에서 요청된 메모리 오퍼레이션은 ACQUIRE 오퍼레이션이 완료된 후에 완료될 수 있습니다.

### (2) RELEASE 오퍼레이션의 영향:

RELEASE 앞에서 요청된 메모리 오퍼레이션은 RELEASE 오퍼레이션이 완료되기 전에 완료됩니다.

RELEASE 뒤에서 요청된 메모리 오퍼레이션은 RELEASE 오퍼레이션 완료 전에 완료될 수 있습니다.

### (3) ACQUIRE vs ACQUIRE 영향:

어떤 ACQUIRE 오퍼레이션보다 앞에서 요청된 모든 ACQUIRE 오퍼레이션은 그 ACQUIRE 오퍼레이션 전에 완료됩니다.

### (4) ACQUIRE vs RELEASE implication:

어떤 RELEASE 오퍼레이션보다 앞서 요청된 ACQUIRE 오퍼레이션은 그 RELEASE 오퍼레이션보다 먼저 완료됩니다.

### (5) 실패한 조건적 ACQUIRE 영향:

ACQUIRE 오퍼레이션의 일부 락(lock) 변종은 락이 곧바로 획득하기에는 불가능한 상태이거나 락이 획득 가능해지도록 기다리는 도중 시그널을 받거나 해서 실패할 수 있습니다. 실패한 락은 어떤 배리어도 내포하지 않습니다.

[!] 참고: 락 ACQUIRE 와 RELEASE 가 단방향 배리어여서 나타나는 현상 중 하나는 크리티컬 섹션 바깥의 인스트럭션의 영향이 크리티컬 섹션 내부로도 들어올 수 있다는 것입니다.

RELEASE 후에 요청되는 ACQUIRE 는 전체 메모리 배리어라 여겨지면 안되는데,

ACQUIRE 앞의 액세스가 ACQUIRE 후에 수행될 수 있고, RELEASE 후의 액세스가 RELEASE 전에 수행될 수도 있으며, 그 두개의 액세스가 서로를 지나칠 수도 있기 때문입니다:

```
*A = a;  
ACQUIRE M  
RELEASE M  
*B = b;
```

는 다음과 같이 될 수도 있습니다:

```
ACQUIRE M, STORE *B, STORE *A, RELEASE M
```

ACQUIRE 와 RELEASE 가 락 획득과 해제라면, 그리고 락의 ACQUIRE 와 RELEASE 가 같은 락 변수에 대한 것이라면, 해당 락을 쥐고 있지 않은 다른 CPU 의 시야에는 이와 같은 재배치가 일어나는 것으로 보일 수 있습니다. 요약하자면, ACQUIRE 에 이어 RELEASE 오퍼레이션을 순차적으로 실행하는 행위가 전체 메모리 배리어로 생각되어선 -안됩니다-.

비슷하게, 앞의 반대 케이스인 RELEASE 와 ACQUIRE 두개 오퍼레이션의 순차적 실행 역시 전체 메모리 배리어를 내포하지 않습니다. 따라서, RELEASE, ACQUIRE 로 규정되는 크리티컬 섹션의 CPU 수행은 RELEASE 와 ACQUIRE 를 가로지를 수 있으므로, 다음과 같은 코드는:

```
*A = a;  
RELEASE M  
ACQUIRE N  
*B = b;
```

다음과 같이 수행될 수 있습니다:

```
ACQUIRE N, STORE *B, STORE *A, RELEASE M
```

이런 재배치는 데드락을 일으킬 수도 있을 것처럼 보일 수 있습니다. 하지만, 그런 데드락의 조짐이 있다면 RELEASE 는 단순히 완료될 것이므로 데드락은 존재할 수 없습니다.

이게 어떻게 올바른 동작을 할 수 있을까요?

우리가 이야기 하고 있는건 재배치를 하는 CPU 에 대한 이야기이지, 컴파일러에 대한 것이 아니란 점이 핵심입니다. 컴파일러 (또는, 개발자) 가 오퍼레이션들을 이렇게 재배치하면, 데드락이 일어날 수 -있습-니다.

하지만 CPU 가 오퍼레이션들을 재배치 했다는걸 생각해 보세요. 이 예에서, 어셈블리 코드 상으로는 언락이 락을 앞서게 되어 있습니다. CPU 가 이를 재배치해서 뒤의 락 오퍼레이션을 먼저 실행하게 됩니다. 만약 데드락이 존재한다면, 이 락 오퍼레이션은 그저 스판을 하며 계속해서 락을 시도합니다 (또는, 한참 후에겠지만, 잠듭니다). CPU 는 언젠가는 (어셈블리 코드에서는 락을 앞서는) 언락 오퍼레이션을 실행하는데, 이 언락 오퍼레이션이 잠재적 데드락을 해결하고, 락 오퍼레이션도 뒤이어 성공하게 됩니다.

하지만 만약 락이 잠을 자는 타입이었다면요? 그런 경우에 코드는 스케줄러로 들어가려 할 거고, 여기서 결국은 메모리 배리어를 만나게 되는데, 이 메모리 배리어는 앞의 언락 오퍼레이션이 완료되도록 만들고, 데드락은 이번에도 해결됩니다. 잠을 자는 행위와 언락 사이의 경주 상황 (race) 도 있을 수 있겠습니다만, 락 관련 기능들은 그런 경주 상황을 모든 경우에 제대로 해결할 수 있어야 합니다.

락과 세마포어는 UP 컴파일된 시스템에서의 순서에 대해 보장을 하지 않기 때문에, 그런 상황에서 인터럽트 비활성화 오퍼레이션과 함께가 아니라면 어떤 일에도 - 특히 I/O 액세스와 관련해서는 - 제대로 사용될 수 없을 겁니다.

"CPU 간 ACQUIRING 배리어 효과" 섹션도 참고하시기 바랍니다.

예를 들어, 다음과 같은 코드를 생각해 봅시다:

```
*A = a;
*B = b;
ACQUIRE
*C = c;
*D = d;
RELEASE
*E = e;
*F = f;
```

여기선 다음의 이벤트 시퀀스가 생길 수 있습니다:

ACQUIRE, {\*F,\*A}, \*E, {\*C,\*D}, \*B, RELEASE

[+] {\*F,\*A} 는 조합된 액세스를 의미합니다.

하지만 다음과 같은 건 불가능하죠:

{*F,*A}, *B,	ACQUIRE, *C, *D,	RELEASE, *E
*A, *B, *C,	ACQUIRE, *D,	RELEASE, *E, *F
*A, *B,	ACQUIRE, *C,	RELEASE, *D, *E, *F
*B,	ACQUIRE, *C, *D,	RELEASE, {*F,*A}, *E

#### 인터럽트 비활성화 함수

인터럽트를 비활성화 하는 함수 (ACQUIRE 와 동일) 와 인터럽트를 활성화 하는 함수 (RELEASE 와 동일) 는 컴파일러 배리어처럼만 동작합니다. 따라서, 별도의 메모리 배리어나 I/O 배리어가 필요한 상황이라면 그 배리어들은 인터럽트 비활성화 함수 외의 방법으로 제공되어야만 합니다.

#### 슬립과 웨이크업 함수

글로벌 데이터에 표시된 이벤트에 의해 프로세스를 잠에 빠트리는 것과 깨우는 것은 해당 이벤트를 기다리는 태스크의 태스크 상태와 그 이벤트를 알리기 위해 사용되는 글로벌 데이터, 두 데이터간의 상호작용으로 볼 수 있습니다. 이것이 옳은 순서대로 일어남을 분명히 하기 위해, 프로세스를 잠에 들게 하는 기능과 깨우는 기능은 몇가지 배리어를 내포합니다.

먼저, 잠을 재우는 쪽은 일반적으로 다음과 같은 이벤트 시퀀스를 따릅니다:

```
for (;;) {
    set_current_state(TASK_UNINTERRUPTIBLE);
    if (event_indicated)
        break;
    schedule();
}
```

`set_current_state()`에 의해, 태스크 상태가 바뀐 후 범용 메모리 배리어가 자동으로 삽입됩니다:

```
CPU 1
=====
set_current_state();
smp_store_mb();
STORE current->state
<범용 배리어>
LOAD event_indicated
```

`set_current_state()`는 다음의 것들로 감싸질 수도 있습니다:

```
prepare_to_wait();
prepare_to_wait_exclusive();
```

이것들 역시 상태를 설정한 후 범용 메모리 배리어를 삽입합니다.  
앞의 전체 시퀀스는 다음과 같은 함수들로 한번에 수행 가능한데, 이것들은 모두 올바른 장소에 메모리 배리어를 삽입합니다:

```
wait_event();
wait_event_interruptible();
wait_event_interruptible_exclusive();
wait_event_interruptible_timeout();
wait_event_killable();
wait_event_timeout();
wait_on_bit();
wait_on_bit_lock();
```

두번째로, 깨우기를 수행하는 코드는 일반적으로 다음과 같을 겁니다:

```
event_indicated = 1;
wake_up(&event_wait_queue);
```

또는:

```
event_indicated = 1;
wake_up_process(event_daemon);
```

wake\_up() 이 무언가를 깨우게 되면, 이 함수는 범용 메모리 배리어를 수행합니다. 이 함수가 아무것도 깨우지 않는다면 메모리 배리어는 수행될 수도, 수행되지 않을 수도 있습니다; 이 경우에 메모리 배리어를 수행할 거라 오해해선 안됩니다. 이 배리어는 태스크 상태가 접근되기 전에 수행되는데, 자세히 말하면 이 이벤트를 알리기 위한 STORE 와 TASK\_RUNNING 으로 상태를 쓰는 STORE 사이에 수행됩니다:

CPU 1 (Sleeper)	CPU 2 (Waker)
<hr/>	
set_current_state();	STORE event_indicated
smp_mb();	wake_up();
STORE current->state	...
<범용 배리어>	<범용 배리어>
LOAD event_indicated	if ((LOAD task->state) & TASK_NORMAL)
	STORE task->state

여기서 "task" 는 깨어나지는 쓰레드이고 CPU 1 의 "current" 와 같습니다.

반복하지만, wake\_up() 이 무언가를 정말 깨운다면 범용 메모리 배리어가 수행될 것이 보장되지만, 그렇지 않다면 그런 보장이 없습니다. 이걸 이해하기 위해, X 와 Y 는 모두 0 으로 초기화 되어 있다는 가정 하에 아래의 이벤트 시퀀스를 생각해 봅시다:

CPU 1	CPU 2
<hr/>	
X = 1;	Y = 1;
smp_mb();	wake_up();
LOAD Y	LOAD X

정말로 깨우기가 행해졌다면, 두 로드 중 (최소한) 하나는 1 을 보게 됩니다.

반면에, 실제 깨우기가 행해지지 않았다면, 두 로드 모두 0을 볼 수도 있습니다.

wake\_up\_process() 는 항상 범용 메모리 배리어를 수행합니다. 이 배리어 역시 태스크 상태가 접근되기 전에 수행됩니다. 특히, 앞의 예제 코드에서 wake\_up() 이 wake\_up\_process() 로 대체된다면 두 로드 중 하나는 1을 볼 것이 보장됩니다.

사용 가능한 깨우기류 함수들로 다음과 같은 것들이 있습니다:

```
complete();
wake_up();
wake_up_all();
wake_up_bit();
wake_up_interruptible();
wake_up_interruptible_all();
wake_up_interruptible_nr();
wake_up_interruptible_poll();
wake_up_interruptible_sync();
wake_up_interruptible_sync_poll();
wake_up_locked();
wake_up_locked_poll();
```

```
wake_up_nr();
wake_up_poll();
wake_up_process();
```

메모리 순서규칙 관점에서, 이 함수들은 모두 `wake_up()` 과 같거나 보다 강한 순서 보장을 제공합니다.

[!] 잠재우는 코드와 깨우는 코드에 내포되는 메모리 배리어들은 깨우기 전에 이루어진 스토어를 잠재우는 코드가 `set_current_state()` 를 호출한 후에 행하는 로드에 대해 순서를 맞추지 \_않는다는\_ 점을 기억하세요. 예를 들어, 잠재우는 코드가 다음과 같고:

```
set_current_state(TASK_INTERRUPTIBLE);
if (event_indicated)
    break;
__set_current_state(TASK_RUNNING);
do_something(my_data);
```

깨우는 코드는 다음과 같다면:

```
my_data = value;
event_indicated = 1;
wake_up(&event_wait_queue);
```

`event_indeacted` 에의 변경이 잠재우는 코드에게 `my_data` 에의 변경 후에 이루어진 것으로 인지될 것이라는 보장이 없습니다. 이런 경우에는 양쪽 코드 모두 각각의 데이터 액세스 사이에 메모리 배리어를 직접 쳐야 합니다. 따라서 앞의 재우는 코드는 다음과 같이:

```
set_current_state(TASK_INTERRUPTIBLE);
if (event_indicated) {
    smp_rmb();
    do_something(my_data);
}
```

그리고 깨우는 코드는 다음과 같이 되어야 합니다:

```
my_data = value;
smp_wmb();
event_indicated = 1;
wake_up(&event_wait_queue);
```

그외의 함수들

-----

그외의 배리어를 내포하는 함수들은 다음과 같습니다:

(\*) `schedule()` 과 그 유사한 것들이 완전한 메모리 배리어를 내포합니다.

## CPU 간 ACQUIRING 배리어의 효과

SMP 시스템에서의 락 기능들은 더욱 강력한 형태의 배리어를 제공합니다: 이 배리어는 동일한 락을 사용하는 다른 CPU 들의 메모리 액세스 순서에도 영향을 끼칩니다.

## ACQUIRE VS 메모리 액세스

다음의 예를 생각해 봅시다: 시스템은 두개의 스핀락 (M) 과 (Q), 그리고 세개의 CPU 를 가지고 있습니다; 여기에 다음의 이벤트 시퀀스가 발생합니다:

CPU 1	CPU 2
=====	=====
WRITE_ONCE(*A, a);	WRITE_ONCE(*E, e);
ACQUIRE M	ACQUIRE Q
WRITE_ONCE(*B, b);	WRITE_ONCE(*F, f);
WRITE_ONCE(*C, c);	WRITE_ONCE(*G, g);
RELEASE M	RELEASE Q
WRITE_ONCE(*D, d);	WRITE_ONCE(*H, h);

\*A 로의 액세스부터 \*H 로의 액세스까지가 어떤 순서로 CPU 3 에게 보여질지에 대해서는 각 CPU에서의 락 사용에 의해 내포되어 있는 제약을 제외하고는 어떤 보장도 존재하지 않습니다. 예를 들어, CPU 3 에게 다음과 같은 순서로 보여지는 것이 가능합니다:

\*E, ACQUIRE M, ACQUIRE Q, \*G, \*C, \*F, \*A, \*B, RELEASE Q, \*D, \*H, RELEASE M

하지만 다음과 같이 보이지는 않을 겁니다:

\*B, \*C or \*D preceding ACQUIRE M  
 \*A, \*B or \*C following RELEASE M  
 \*F, \*G or \*H preceding ACQUIRE Q  
 \*E, \*F or \*G following RELEASE Q

## 메모리 배리어가 필요한 곳

설령 SMP 커널을 사용하더라도 싱글 쓰레드로 동작하는 코드는 올바르게 동작하는 것으로 보여질 것이기 때문에, 평범한 시스템 운영중에 메모리 오퍼레이션 재배치는 일반적으로 문제가 되지 않습니다. 하지만, 재배치가 문제가 될 수 있는 네가지 환경이 있습니다:

(\*) 프로세서간 상호 작용.

(\*) 어토믹 오퍼레이션.

(\*) 디바이스 액세스.

(\*) 인터럽트.

### 프로세서간 상호 작용

---

두개 이상의 프로세서를 가진 시스템이 있다면, 시스템의 두개 이상의 CPU 는 동시에 같은 데이터에 대한 작업을 할 수 있습니다. 이는 동기화 문제를 일으킬 수 있고, 이 문제를 해결하는 일반적 방법은 락을 사용하는 것입니다. 하지만, 락은 상당히 비용이 비싸서 가능하면 락을 사용하지 않고 일을 처리하는 것이 낫습니다. 이런 경우, 두 CPU 모두에 영향을 끼치는 오퍼레이션들은 오동작을 막기 위해 신중하게 순서가 맞춰져야 합니다.

예를 들어, R/W 세마포어의 느린 수행경로 (slow path) 를 생각해 봅시다. 세마포어를 위해 대기를 하는 하나의 프로세스가 자신의 스택 중 일부를 이 세마포어의 대기 프로세스 리스트에 링크한 채로 있습니다:

```
struct rw_semaphore {  
    ...  
    spinlock_t lock;  
    struct list_head waiters;  
};  
  
struct rwsem_waiter {  
    struct list_head list;  
    struct task_struct *task;  
};
```

특정 대기 상태 프로세스를 깨우기 위해, up\_read() 나 up\_write() 함수는 다음과 같은 일을 합니다:

- (1) 다음 대기 상태 프로세스 레코드는 어디있는지 알기 위해 이 대기 상태 프로세스 레코드의 next 포인터를 읽습니다;
- (2) 이 대기 상태 프로세스의 task 구조체로의 포인터를 읽습니다;
- (3) 이 대기 상태 프로세스가 세마포어를 획득했음을 알리기 위해 task 포인터를 초기화 합니다;
- (4) 해당 태스크에 대해 wake\_up\_process() 를 호출합니다; 그리고
- (5) 해당 대기 상태 프로세스의 task 구조체를 잡고 있던 레퍼런스를 해제합니다.

달리 말하자면, 다음 이벤트 시퀀스를 수행해야 합니다:

```
LOAD waiter->list.next;  
LOAD waiter->task;  
STORE waiter->task;  
CALL wakeup  
RELEASE task
```

그리고 이 이벤트들이 다른 순서로 수행된다면, 오동작이 일어날 수 있습니다.

한번 세마포어의 대기줄에 들어갔고 세마포어 락을 놓았다면, 해당 대기 프로세스는 락을 다시는 잡지 않습니다; 대신 자신의 task 포인터가 초기화 되길 기다립니다. 그 레코드는 대기 프로세스의 스택에 있기 때문에, 리스트의 next 포인터가 읽혀지기 전에 task 포인터가 지워진다면, 다른 CPU 는 해당 대기 프로세스를 시작해 버리고 up\*() 함수가 next 포인터를 읽기 전에 대기 프로세스의 스택을 마구 건드릴 수 있습니다.

그렇게 되면 위의 이벤트 시퀀스에 어떤 일이 일어나는지 생각해 보죠:

CPU 1	CPU 2
=====	=====
	down_xxx()
	Queue waiter
	Sleep
up_yyy()	
LOAD waiter->task;	
STORE waiter->task;	
<preempt>	Woken up by other event
	Resume processing
	down_xxx() returns
	call foo()
	foo() clobbers *waiter
</preempt>	
LOAD waiter->list.next;	
--- OOPS ---	

이 문제는 세마포어 락의 사용으로 해결될 수도 있겠지만, 그렇게 되면 깨어난 후에 down\_xxx() 함수가 불필요하게 스팬락을 또다시 얻어야만 합니다.

이 문제를 해결하는 방법은 범용 SMP 메모리 배리어를 추가하는 겁니다:

```
LOAD waiter->list.next;
LOAD waiter->task;
smp_mb();
STORE waiter->task;
CALL wakeup
RELEASE task
```

이 경우에, 배리어는 시스템의 나머지 CPU 들에게 모든 배리어 앞의 메모리 액세스가 배리어 뒤의 메모리 액세스보다 앞서 일어난 것으로 보이게 만듭니다. 배리어 앞의 메모리 액세스들이 배리어 명령 자체가 완료되는 시점까지 완료된다고는 보장하지 않습니다.

(이게 문제가 되지 않을) 단일 프로세서 시스템에서 smp\_mb() 는 실제로는 그저 컴파일러가 CPU 안에서의 순서를 바꾸거나 하지 않고 주어진 순서대로 명령을 내리도록 하는 컴파일러 배리어일 뿐입니다. 오직 하나의 CPU 만 있으니, CPU 의 의존성 순서로직이 그 외의 모든것을 알아서 처리할 겁니다.

### 어토믹 오퍼레이션

---

어토믹 오퍼레이션은 기술적으로 프로세서간 상호작용으로 분류되며 그 종 일부는 전체 메모리 배리어를 내포하고 또 일부는 내포하지 않지만, 커널에서 상당히 의존적으로 사용하는 기능 중 하나입니다.

더 많은 내용을 위해선 [Documentation/atomic\\_t.txt](#) 를 참고하세요.

### 디바이스 액세스

---

많은 디바이스가 메모리 매핑 기법으로 제어될 수 있는데, 그렇게 제어되는 디바이스는 CPU 에는 단지 특정 메모리 영역의 집합처럼 보이게 됩니다. 드라이버는 그런 디바이스를 제어하기 위해 정확히 올바른 순서로 올바른 메모리 액세스를 만들어야 합니다.

하지만, 액세스들을 재배치하거나 조합하거나 병합하는게 더 효율적이라 판단하는 영리한 CPU 나 컴파일러들을 사용하면 드라이버 코드의 조심스럽게 순서 맞춰진 액세스들이 디바이스에는 요청된 순서대로 도착하지 못하게 할 수 있는 - 디바이스가 오동작을 하게 할 - 잠재적 문제가 생길 수 있습니다.

리눅스 커널 내부에서, I/O 는 어떻게 액세스들을 적절히 순차적이게 만들 수 있는지 알고 있는, - `inb()` 나 `writel()` 과 같은 - 적절한 액세스 루틴을 통해 이루어져야만 합니다. 이것들은 대부분의 경우에는 명시적 메모리 배리어 와 함께 사용될 필요가 없습니다만, 완화된 메모리 액세스 속성으로 I/O 메모리 원도우로의 참조를 위해 액세스 함수가 사용된다면 순서를 강제하기 위해 `_mandatory_` 메모리 배리어가 필요합니다.

더 많은 정보를 위해선 [Documentation/driver-api/device-io.rst](#) 를 참고하십시오.

### 인터럽트

---

드라이버는 자신의 인터럽트 서비스 루틴에 의해 인터럽트 당할 수 있기 때문에 드라이버의 이 두 부분은 서로의 디바이스 제어 또는 액세스 부분과 상호 간섭할 수 있습니다.

스스로에게 인터럽트 당하는 걸 불가능하게 하고, 드라이버의 크리티컬한 오퍼레이션들을 모두 인터럽트가 불가능하게 된 영역에 집어넣거나 하는 방법 (락의 한 형태) 으로 이런 상호 간섭을 - 최소한 부분적으로라도 - 줄일 수 있습니다. 드라이버의 인터럽트 루틴이 실행 중인 동안, 해당 드라이버의 코어는 같은 CPU 에서 수행되지 않을 것이며, 현재의 인터럽트가 처리되는 중에는 또다시 인터럽트가 일어나지 못하도록 되어 있으니 인터럽트 핸들러는 그에 대해서는 락을 잡지 않아도 됩니다.

하지만, 어드레스 레지스터와 데이터 레지스터를 갖는 이더넷 카드를 다루는 드라이버를 생각해 봅시다. 만약 이 드라이버의 코어가 인터럽트를 비활성화시킨 채로 이더넷 카드와 대화하고 드라이버의 인터럽트 핸들러가 호출되었다면:

```

LOCAL IRQ DISABLE
writew(ADDR, 3);
writew(DATA, y);
LOCAL IRQ ENABLE
<interrupt>
writew(ADDR, 4);
q = readw(DATA);
</interrupt>

```

만약 순서 규칙이 충분히 완화되어 있다면 데이터 레지스터에의 스토어는 어드레스 레지스터에 두번째로 행해지는 스토어 뒤에 일어날 수도 있습니다:

```
STORE *ADDR = 3, STORE *ADDR = 4, STORE *DATA = y, q = LOAD *DATA
```

만약 순서 규칙이 충분히 완화되어 있고 묵시적으로든 명시적으로든 배리어가 사용되지 않았다면 인터럽트 비활성화 섹션에서 일어난 액세스가 바깥으로 새어서 인터럽트 내에서 일어난 액세스와 섞일 수 있다고 - 그리고 그 반대도 - 가정해야만 합니다.

그런 영역 안에서 일어나는 I/O 액세스는 묵시적 I/O 배리어를 형성하는, 엄격한 순서 규칙의 I/O 레지스터로의 로드 오퍼레이션을 포함하기 때문에 일반적으로는 문제가 되지 않습니다.

하나의 인터럽트 루틴과 별도의 CPU에서 수행중이며 서로 통신을 하는 두 루틴 사이에도 비슷한 상황이 일어날 수 있습니다. 만약 그런 경우가 발생할 가능성이 있다면, 순서를 보장하기 위해 인터럽트 비활성화 락이 사용되어져야만 합니다.

---

### =====

#### 커널 I/O 배리어의 효과

---

I/O 액세스를 통한 주변장치와의 통신은 아키텍쳐와 기기에 매우 종속적입니다. 따라서, 본질적으로 이식성이 없는 드라이버는 가능한 가장 적은 오버헤드로 동기화를 하기 위해 각자의 타겟 시스템의 특정 동작에 의존할 겁니다. 다양한 아키텍처와 버스 구현에 이식성을 가지려 하는 드라이버를 위해, 커널은 다양한 정도의 순서 보장을 제공하는 일련의 액세스 함수를 제공합니다.

(\*) `readX()`, `writeX()`:

`readX()` 와 `writeX()` MMIO 액세스 함수는 접근되는 주변장치로의 포인터를 `_iomem *` 패러미터로 받습니다. 디폴트 I/O 기능으로 매핑되는 포인터 (예: `ioremap()` 으로 반환되는 것) 의 순서 보장은 다음과 같습니다:

1. 같은 주변장치로의 모든 `readX()` 와 `writeX()` 액세스는 각자에 대해 순서지어집니다. 이는 같은 CPU 쓰레드에 의한 특정 디바이스로의 MMIO 레지스터 액세스가 프로그램 순서대로 도착할 것을 보장합니다.
2. 한 스피너락을 잡은 CPU 쓰레드에 의한 `writeX()` 는 같은 스피너락을 나중에 잡은 다른 CPU 쓰레드에 의해 같은 주변장치를 향해 호출된 `writeX()`

앞으로 순서지어집니다. 이는 스피너를 잡은 채 특정 디바이스를 향해  
호출된 MMIO 레지스터 쓰기는 해당 락의 획득에 일관적인 순서로 도달할  
것을 보장합니다.

3. 특정 주변장치를 향한 특정 CPU 쓰레드의 `writeX()` 는 먼저 해당 쓰레드로 전파되는, 또는 해당 쓰레드에 의해 요청된 모든 앞선 메모리 쓰기가 완료되기 전까지 먼저 기다립니다. 이는 `dma_alloc_coherent()` 를 통해 할당된 전송용 DMA 버퍼로의 해당 CPU 의 쓰기가 이 CPU 가 이 전송을 시작시키기 위해 MMIO 컨트롤 레지스터에 쓰기를 할 때 DMA 엔진에 보여질 것을 보장합니다.
4. 특정 CPU 쓰레드에 의한 주변장치로의 `readX()` 는 같은 쓰레드에 의한 모든 뒤따르는 메모리 읽기가 시작되기 전에 완료됩니다. 이는 `dma_alloc_coherent()` 를 통해 할당된 수신용 DMA 버퍼로부터의 CPU 의 읽기는 이 DMA 수신의 완료를 표시하는 DMA 엔진의 MMIO 상태 레지스터 읽기 후에는 오염된 데이터를 읽지 않을 것을 보장합니다.
5. CPU 에 의한 주변장치로의 `readX()` 는 모든 뒤따르는 `delay()` 루프가 수행을 시작하기 전에 완료됩니다. 이는 CPU 의 특정 주변장치로의 두개의 MMIO 레지스터 쓰기가 행해지는데 첫번째 쓰기가 `readX()` 를 통해 곧바로 읽어졌고 이어 두번째 `writeX()` 전에 `udelay(1)` 이 호출되었다면 이 두개의 쓰기는 최소 1us 의 간격을 두고 행해질 것을 보장합니다:

```
writel(42, DEVICE_REGISTER_0); // 디바이스에 도착함...
readl(DEVICE_REGISTER_0);
udelay(1);
writel(42, DEVICE_REGISTER_1); // ...이것보다 최소 1us 전에.
```

디플트가 아닌 기능을 통해 얻어지는 `__iomem` 포인터 (예: `ioremap_wc()` 를 통해 리턴되는 것) 의 순서 속성은 실제 아키텍쳐에 의존적이어서 이런 종류의 매핑으로의 액세스는 앞서 설명된 보장사항에 의존할 수 없습니다.

(\*) `readX_relaxed()`, `writeX_relaxed()`

이것들은 `readX()` 와 `writeX()` 랑 비슷하지만, 더 완화된 메모리 순서 보장을 제공합니다. 구체적으로, 이것들은 일반적 메모리 액세스나 `delay()` 루프 (예: 앞의 2-5 항목) 에 대해 순서를 보장하지 않습니다만 디플트 I/O 기능으로 매핑된 `__iomem` 포인터에 대해 동작할 때, 같은 CPU 쓰레드에 의한 같은 주변장치로의 액세스에는 순서가 맞춰질 것이 보장됩니다.

(\*) `readsX()`, `writesX()`:

`readsX()` 와 `writesX()` MMIO 액세스 함수는 DMA 를 수행하는데 적절치 않은, 주변장치 내의 메모리 매핑된 레지스터 기반 FIFO 로의 액세스를 위해 설계되었습니다. 따라서, 이 기능들은 앞서 설명된 `readX_relaxed()` 와 `writeX_relaxed()` 의 순서 보장만을 제공합니다.

(\*) `inX()`, `outX()`:

`inX()` 와 `outX()` 액세스 함수는 일부 아키텍쳐 (특히 x86) 에서는 특수한 명령어를 필요로 하며 포트에 매핑되는, 과거의 유산인 I/O 주변장치로의

접근을 위해 만들어졌습니다.

많은 CPU 아키텍쳐가 결국은 이런 주변장치를 내부의 가상 메모리 맵핑을 통해 접근하기 때문에, `inX()` 와 `outX()` 가 제공하는 이식성 있는 순서 보장은 디폴트 I/O 기능을 통한 맵핑을 접근할 때의 `readX()` 와 `writeX()` 에 의해 제공되는 것과 각각 동일합니다.

디바이스 드라이버는 `outX()` 가 리턴하기 전에 해당 I/O 주변장치로부터의 완료 응답을 기다리는 쓰기 트랜잭션을 만들어 낸다고 기대할 수도 있습니다. 이는 모든 아키텍쳐에서 보장되지는 않고, 따라서 이식성 있는 순서 규칙의 일부분이 아닙니다.

#### (\*) `insX()`, `outsX()`:

앞에서와 같이, `insX()` 와 `outsX()` 액세스 함수는 디폴트 I/O 기능을 통한 맵핑을 접근할 때 각각 `readX()` 와 `writeX()` 와 같은 순서 보장을 제공합니다.

#### (\*) `ioreadX()`, `iowriteX()`

이것들은 `inX()/outX()` 나 `readX()/writeX()` 처럼 실제로 수행하는 액세스의 종류에 따라 적절하게 수행될 것입니다.

`String` 액세스 함수 (`insX()`, `outsX()`, `readsX()` 그리고 `writesX()`) 의 예외를 제외하고는, 앞의 모든 것이 아랫단의 주변장치가 little-endian 이라 가정하며, 따라서 big-endian 아키텍쳐에서는 byte-swapping 오퍼레이션을 수행합니다.

---

#### ===== 가정되는 가장 완화된 실행 순서 모델 =====

컨셉적으로 CPU 는 주어진 프로그램에 대해 프로그램 그 자체에는 인과성 (program causality) 을 지키는 것처럼 보이게 하지만 일반적으로는 순서를 거의 지켜주지 않는다고 가정되어야만 합니다. (i386 이나 x86\_64 같은) 일부 CPU 들은 코드 재배치에 (powerpc 나 frv 와 같은) 다른 것들에 비해 강한 제약을 갖지만, 아키텍처 종속적 코드 이외의 코드에서는 순서에 대한 제약이 가장 완화된 경우 (DEC Alpha) 를 가정해야 합니다.

이 말은, CPU 에게 주어지는 인스트럭션 스트림 내의 한 인스트럭션이 앞의 인스트럭션에 종속적이라면 앞의 인스트럭션은 뒤의 종속적 인스트럭션이 실행되기 전에 완료[\*]될 수 있어야 한다는 제약 (달리 말해서, 인과성이 지켜지는 것으로 보이게 함) 외에는 자신이 원하는 순서대로 - 심지어 병렬적으로도 - 그 스트림을 실행할 수 있음을 의미합니다

[\*] 일부 인스트럭션은 하나 이상의 영향 - 조건 코드를 바꾼다던지, 레지스터나 메모리를 바꾼다던지 - 을 만들어내며, 다른 인스트럭션은 다른 효과에 종속적일 수 있습니다.

CPU 는 최종적으로 아무 효과도 만들지 않는 인스트럭션 시퀀스는 없애버릴 수도 있습니다. 예를 들어, 만약 두개의 연속되는 인스트럭션이 둘 다 같은 레지스터에 직접적인 값 (immediate value) 을 집어넣는다면, 첫번째 인스트럭션은 버려질 수도

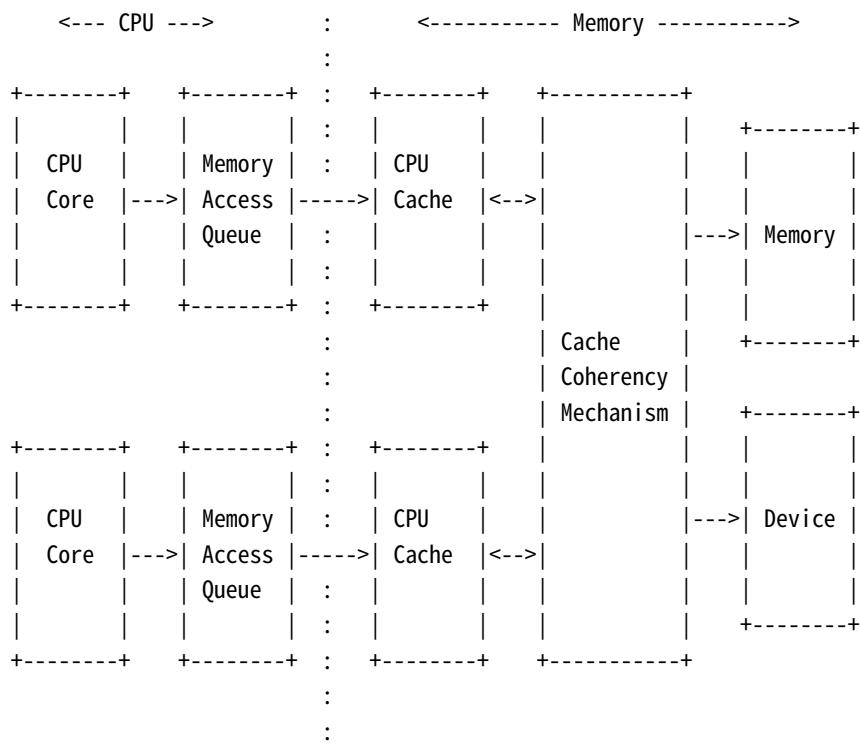
있습니다.

비슷하게, 컴파일러 역시 프로그램의 인과성만 지켜준다면 인스트럭션 스트림을 자신이 보기에 올바르다 생각되는대로 재배치 할 수 있습니다.

### CPU 캐시의 영향

캐시된 메모리 오퍼레이션들이 시스템 전체에 어떻게 인지되는지는 CPU 와 메모리 사이에 존재하는 캐시들, 그리고 시스템 상태의 일관성을 관리하는 메모리 일관성 시스템에 상당 부분 영향을 받습니다.

한 CPU 가 시스템의 다른 부분들과 캐시를 통해 상호작용한다면, 메모리 시스템은 CPU 의 캐시들을 포함해야 하며, CPU 와 CPU 자신의 캐시 사이에서의 동작을 위한 메모리 배리어를 가져야 합니다. (메모리 배리어는 논리적으로는 다음 그림의 점선에서 동작합니다):



특정 로드나 스토어는 해당 오퍼레이션을 요청한 CPU 의 캐시 내에서 동작을 완료할 수도 있기 때문에 해당 CPU 의 바깥에는 보이지 않을 수 있지만, 다른 CPU 가 관심을 갖는다면 캐시 일관성 메커니즘이 해당 캐시라인을 해당 CPU 에게 전달하고, 해당 메모리 영역에 대한 오퍼레이션이 발생할 때마다 그 영향을 전파시키기 때문에, 해당 오퍼레이션은 메모리에 실제로 액세스를 한것처럼 나타날 것입니다.

CPU 코어는 프로그램의 인과성이 유지된다고만 여겨진다면 인스트럭션들을 어떤 순서로든 재배치해서 수행할 수 있습니다. 일부 인스트럭션들은 로드나 스토어 오퍼레이션을 만드는데 이 오퍼레이션들은 이후 수행될 메모리 액세스 큐에 들어가게 됩니다. 코어는 이 오퍼레이션들을 해당 큐에 어떤 순서로든 원하는대로 넣을 수

있고, 다른 인스트럭션의 완료를 기다리도록 강제되기 전까지는 수행을 계속합니다.

메모리 배리어가 하는 일은 CPU 쪽에서 메모리 쪽으로 넘어가는 액세스들의 순서, 그리고 그 액세스의 결과가 시스템의 다른 관찰자들에게 인지되는 순서를 제어하는 것입니다.

[!] CPU 들은 항상 그들 자신의 로드와 스토어는 프로그램 순서대로 일어난 것으로 보기 때문에, 주어진 CPU 내에서는 메모리 배리어를 사용할 필요가 \_없습니다\_.

[!] MMIO 나 다른 디바이스 액세스들은 캐시 시스템을 우회할 수도 있습니다. 우회 여부는 디바이스가 액세스 되는 메모리 윈도우의 특성에 의해 결정될 수도 있고, CPU 가 가지고 있을 수 있는 특수한 디바이스 통신 인스트럭션의 사용에 의해서 결정될 수도 있습니다.

#### 캐시 일관성 VS DMA

---

모든 시스템이 DMA 를 하는 디바이스에 대해서까지 캐시 일관성을 유지하지는 않습니다. 그런 경우, DMA 를 시도하는 디바이스는 RAM 으로부터 잘못된 데이터를 읽을 수 있는데, 더티 캐시 라인이 CPU 의 캐시에 머무르고 있고, 바뀐 값이 아직 RAM 에 써지지 않았을 수 있기 때문입니다. 이 문제를 해결하기 위해선, 커널의 적절한 부분에서 각 CPU 캐시의 문제되는 비트들을 플러시 (flush) 시켜야만 합니다 (그리고 그것들을 무효화 - invalidation - 시킬 수도 있겠죠).

또한, 디바이스에 의해 RAM 에 DMA 로 쓰여진 값은 디바이스가 쓰기를 완료한 후에 CPU 의 캐시에서 RAM 으로 쓰여지는 더티 캐시 라인에 의해 덮어써질 수도 있고, CPU 의 캐시에 존재하는 캐시 라인이 해당 캐시에서 삭제되고 다시 값을 읽어들이기 전까지는 RAM 이 업데이트 되었다는 사실 자체가 숨겨져 버릴 수도 있습니다. 이 문제를 해결하기 위해선, 커널의 적절한 부분에서 각 CPU 의 캐시 안의 문제가 되는 비트들을 무효화 시켜야 합니다.

캐시 관리에 대한 더 많은 정보를 위해선 Documentation/core-api/cachetlb.rst 를 참고하세요.

#### 캐시 일관성 VS MMIO

---

Memory mapped I/O 는 일반적으로 CPU 의 메모리 공간 내의 한 윈도우의 특정 부분 내의 메모리 지역에 이루어지는데, 이 윈도우는 일반적인, RAM 으로 향하는 윈도우와는 다른 특성을 갖습니다.

그런 특성 가운데 하나는, 일반적으로 그런 액세스는 캐시를 완전히 우회하고 디바이스 버스로 곧바로 향한다는 것입니다. 이 말은 MMIO 액세스는 먼저 시작되어서 캐시에서 완료된 메모리 액세스를 추월할 수 있다는 뜻입니다. 이런 경우엔 메모리 배리어만으로는 충분치 않고, 만약 캐시된 메모리 쓰기 오퍼레이션과 MMIO 액세스가 어떤 방식으로든 의존적이라면 해당 캐시는 두 오퍼레이션 사이에 비워져(flush)야만 합니다.

CPU 들이 저지르는 일들

---

프로그래머는 CPU 가 메모리 오퍼레이션들을 정확히 요청한대로 수행해 줄 것이라고 생각하는데, 예를 들어 다음과 같은 코드를 CPU 에게 넘긴다면:

```
a = READ_ONCE(*A);
WRITE_ONCE(*B, b);
c = READ_ONCE(*C);
d = READ_ONCE(*D);
WRITE_ONCE(*E, e);
```

CPU 는 다음 인스트럭션을 처리하기 전에 현재의 인스트럭션을 위한 메모리 오퍼레이션을 완료할 것이라 생각하고, 따라서 시스템 외부에서 관찰하기에도 정해진 순서대로 오퍼레이션이 수행될 것으로 예상합니다:

```
LOAD *A, STORE *B, LOAD *C, LOAD *D, STORE *E.
```

당연하지만, 실제로는 훨씬 엉망입니다. 많은 CPU 와 컴파일러에서 앞의 가정은 성립하지 못하는데 그 이유는 다음과 같습니다:

- (\*) 로드 오퍼레이션들은 실행을 계속 해나가기 위해 곧바로 완료될 필요가 있는 경우가 많은 반면, 스토어 오퍼레이션들은 종종 별다른 문제 없이 유예될 수 있습니다;
- (\*) 로드 오퍼레이션들은 예측적으로 수행될 수 있으며, 필요없는 로드였다고 증명된 예측적 로드의 결과는 버려집니다;
- (\*) 로드 오퍼레이션들은 예측적으로 수행될 수 있으므로, 예상된 이벤트의 시퀀스와 다른 시간에 로드가 이뤄질 수 있습니다;
- (\*) 메모리 액세스 순서는 CPU 버스와 캐시를 좀 더 잘 사용할 수 있도록 재배치 될 수 있습니다;
- (\*) 로드와 스토어는 인접한 위치에의 액세스들을 일괄적으로 처리할 수 있는 메모리나 I/O 하드웨어 (메모리와 PCI 디바이스 둘 다 이게 가능할 수 있습니다) 에 대해 요청되는 경우, 개별 오퍼레이션을 위한 트랜잭션 설정 비용을 아끼기 위해 조합되어 실행될 수 있습니다; 그리고
- (\*) 해당 CPU 의 데이터 캐시가 순서에 영향을 끼칠 수도 있고, 캐시 일관성 메커니즘이 - 스토어가 실제로 캐시에 도달한다면 - 이 문제를 완화시킬 수는 있지만 이 일관성 관리가 다른 CPU 들에도 같은 순서로 전달된다는 보장은 없습니다.

따라서, 앞의 코드에 대해 다른 CPU 가 보는 결과는 다음과 같을 수 있습니다:

```
LOAD *A, ..., LOAD {*C,*D}, STORE *E, STORE *B
```

("LOAD {\*C,\*D}" 는 조합된 로드입니다)

하지만, CPU 는 스스로는 일관적일 것을 보장합니다: CPU \_자신\_ 의 액세스들은 자신에게는 메모리 배리어가 없음에도 불구하고 정확히 순서 세워진 것으로 보여질 것입니다. 예를 들어 다음의 코드가 주어졌다면:

```
U = READ_ONCE(*A);
WRITE_ONCE(*A, V);
WRITE_ONCE(*A, W);
X = READ_ONCE(*A);
WRITE_ONCE(*A, Y);
Z = READ_ONCE(*A);
```

그리고 외부의 영향에 의한 간섭이 없다고 가정하면, 최종 결과는 다음과 같이 나타날 것이라고 예상될 수 있습니다:

```
U == *A 의 최초 값
X == W
Z == Y
*A == Y
```

앞의 코드는 CPU 가 다음의 메모리 액세스 시퀀스를 만들도록 할겁니다:

```
U=LOAD *A, STORE *A=V, STORE *A=W, X=LOAD *A, STORE *A=Y, Z=LOAD *A
```

하지만, 별다른 개입이 없고 프로그램의 시야에 이 세상이 여전히 일관적이라고 보인다는 보장만 지켜진다면 이 시퀀스는 어떤 조합으로든 재구성될 수 있으며, 각 액세스들은 합쳐지거나 버려질 수 있습니다. 일부 아키텍처에서 CPU 는 같은 위치에 대한 연속적인 로드 오퍼레이션들을 재배치 할 수 있기 때문에 앞의 예에서의 READ\_ONCE() 와 WRITE\_ONCE() 는 반드시 존재해야 함을 알아두세요. 그런 종류의 아키텍처에서 READ\_ONCE() 와 WRITE\_ONCE() 는 이 문제를 막기 위해 필요한 일을 뭐가 됐든지 하게 되는데, 예를 들어 Itanium 에서는 READ\_ONCE() 와 WRITE\_ONCE() 가 사용하는 volatile 캐스팅은 GCC 가 그런 재배치를 방지하는 특수 인스트럭션인 ld.acq 와 stl.rel 인스트럭션을 각각 만들어 내도록 합니다.

컴파일러 역시 이 시퀀스의 액세스들을 CPU 가 보기도 전에 합치거나 버리거나 뒤로 미뤄버릴 수 있습니다.

예를 들어:

```
*A = V;
*A = W;
```

는 다음과 같이 변형될 수 있습니다:

```
*A = W;
```

따라서, 쓰기 배리어나 WRITE\_ONCE() 가 없다면 \*A 로의 V 값의 저장의 효과는 사라진다고 가정될 수 있습니다. 비슷하게:

```
*A = Y;
Z = *A;
```

는, 메모리 배리어나 READ\_ONCE() 와 WRITE\_ONCE() 없이는 다음과 같이 변형될 수

있습니다:

```
*A = Y;  
Z = Y;
```

그리고 이 LOAD 오퍼레이션은 CPU 바깥에는 아예 보이지 않습니다.

그리고, ALPHA 가 있다

---

DEC Alpha CPU 는 가장 완화된 메모리 순서의 CPU 중 하나입니다. 뿐만 아니라, Alpha CPU 의 일부 버전은 분할된 데이터 캐시를 가지고 있어서, 의미적으로 관계되어 있는 두개의 캐시 라인이 서로 다른 시간에 업데이트 되는게 가능합니다. 이게 데이터 의존성 배리어가 정말 필요해지는 부분인데, 데이터 의존성 배리어는 메모리 일관성 시스템과 함께 두개의 캐시를 동기화 시켜서, 포인터 변경과 새로운 데이터의 발견을 올바른 순서로 일어나게 하기 때문입니다.

리눅스 커널의 메모리 배리어 모델은 Alpha 에 기초해서 정의되었습니다만, v4.15 부터는 Alpha 용 READ\_ONCE() 코드 내에 smp\_mb() 가 추가되어서 메모리 모델로의 Alpha 의 영향력이 크게 줄어들었습니다.

가상 머신 게스트

---

가상 머신에서 동작하는 게스트들은 게스트 자체는 SMP 지원 없이 컴파일 되었다 해도 SMP 영향을 받을 수 있습니다. 이건 UP 커널을 사용하면서 SMP 호스트와 결부되어 발생하는 부작용입니다. 이 경우에는 mandatory 배리어를 사용해서 문제를 해결할 수 있겠지만 그런 해결은 대부분의 경우 최적의 해결책이 아닙니다.

이 문제를 완벽하게 해결하기 위해, 로우 레벨의 virt\_mb() 등의 매크로를 사용할 수 있습니다. 이것들은 SMP 가 활성화 되어 있다면 smp\_mb() 등과 동일한 효과를 갖습니다만, SMP 와 SMP 아닌 시스템 모두에 대해 동일한 코드를 만들어냅니다. 예를 들어, 가상 머신 게스트들은 (SMP 일 수 있는) 호스트와 동기화를 할 때에는 smp\_mb() 가 아니라 virt\_mb() 를 사용해야 합니다.

이것들은 smp\_mb() 류의 것들과 모든 부분에서 동일하며, 특히, MMIO 의 영향에 대해서는 간여하지 않습니다: MMIO 의 영향을 제어하려면, mandatory 배리어를 사용하시기 바랍니다.

=====

사용 예

=====

순환식 버퍼

---

메모리 배리어는 순환식 버퍼를 생성자(producer)와 소비자(consumer) 사이의 동기화에 락을 사용하지 않고 구현하는데에 사용될 수 있습니다. 더 자세한 내용을 위해선 다음을 참고하세요:

Documentation/core-api/circular-buffers.rst

=====

참고 문헌

=====

Alpha AXP Architecture Reference Manual, Second Edition (Sites & Witek, Digital Press)

- Chapter 5.2: Physical Address Space Characteristics
- Chapter 5.4: Caches and Write Buffers
- Chapter 5.5: Data Sharing
- Chapter 5.6: Read/Write Ordering

AMD64 Architecture Programmer's Manual Volume 2: System Programming

- Chapter 7.1: Memory-Access Ordering
- Chapter 7.4: Buffering and Combining Memory Writes

ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile)

- Chapter B2: The AArch64 Application Level Memory Model

IA-32 Intel Architecture Software Developer's Manual, Volume 3:  
System Programming Guide

- Chapter 7.1: Locked Atomic Operations
- Chapter 7.2: Memory Ordering
- Chapter 7.4: Serializing Instructions

The SPARC Architecture Manual, Version 9

- Chapter 8: Memory Models
- Appendix D: Formal Specification of the Memory Models
- Appendix J: Programming with the Memory Models

Storage in the PowerPC (Stone and Fitzgerald)

UltraSPARC Programmer Reference Manual

- Chapter 5: Memory Accesses and Cacheability
- Chapter 15: Sparc-V9 Memory Models

UltraSPARC III Cu User's Manual

- Chapter 9: Memory Models

UltraSPARC IIIf Processor User's Manual

- Chapter 8: Memory Models

UltraSPARC Architecture 2005

- Chapter 9: Memory
- Appendix D: Formal Specifications of the Memory Models

UltraSPARC T1 Supplement to the UltraSPARC Architecture 2005

- Chapter 8: Memory Models
- Appendix F: Caches and Cache Coherency

Solaris Internals, Core Kernel Architecture, p63-68:

Chapter 3.3: Hardware Considerations for Locks and  
Synchronization

Unix Systems for Modern Architectures, Symmetric Multiprocessing and Caching  
for Kernel Programmers:

Chapter 13: Other Memory Models

Intel Itanium Architecture Software Developer's Manual: Volume 1:

Section 2.6: Speculation  
Section 4.4: Memory Access

## **日本語訳**

NOTE: This is a version of Documentation/process/howto.rst translated into Japanese. This document is maintained by Tsugikazu Shibata <[tshibata@ab.jp.nec.com](mailto:tshibata@ab.jp.nec.com)> If you find any difference between this document and the original file or a problem with the translation, please contact the maintainer of this file.

Please also note that the purpose of this file is to be easier to read for non English (read: Japanese) speakers and is not intended as a fork. So if you have any comments or updates for this file, please try to update the original English file first.

---

この文書は、Documentation/process/howto.rst の和訳です。

翻訳者：Tsugikazu Shibata <[tshibata@ab.jp.nec.com](mailto:tshibata@ab.jp.nec.com)>

---

### **\* Linux カーネル開発のやり方**

これは上のトピック (Linux カーネル開発のやり方) の重要な事柄を網羅したドキュメントです。ここには Linux カーネル開発者になるための方法と Linux カーネル開発コミュニティと共に活動するやり方を学ぶ方法が含まれています。カーネルプログラミングに関する技術的な項目に関することは何も含めないようにしていますが、カーネル開発者となるための正しい方向に向かう手助けになります。

もし、このドキュメントのどこかが古くなっていた場合には、このドキュメントの最後にリストしたメンテナにパッチを送ってください。

### **\* はじめに**

あなたは Linux カーネルの開発者になる方法を学びたいのでしょうか？ それとも上司から「このデバイスの Linux ドライバを書くように」と言われたのかもしれません。この文書の目的は、あなたが踏るべき手順と、コミュニティと一緒にうまく働くヒントを書き下すことで、あなたが知るべき全てのことを教えることです。また、このコミュニティがなぜ今うまくまわっているのかという理由も説明しようと試みています。

カーネルは少量のアーキテクチャ依存部分がアセンブリ言語で書かれている以外の大部分は C 言語で書かれています。C 言語をよく理解していることはカーネル開発に必要です。低レベルのアーキテクチャ開発を

するのでなければ、(どんなアーキテクチャでも) アセンブリ (訳注: 言語) は必要ありません。以下の本は、C 言語の十分な知識や何年もの経験に取って代わるものではありませんが、少なくともリファレンスとしては良い本です。

- “The C Programming Language” by Kernighan and Ritchie [Prentice Hall]
- 『プログラミング言語C第2版』(B.W. カーニハン/D.M. リッチャー著石田晴久訳) [共立出版]
- “Practical C Programming” by Steve Oualline [O'Reilly]
- 『C 実践プログラミング第3版』(Steve Oualline著望月康司監訳谷口功訳) [オライリージャパン]
- “C: A Reference Manual” by Harbison and Steele [Prentice Hall]
- 『新・詳説 C 言語 H&S リファレンス』(サミュエル P ハービソン/ガイ L スティール共著齊藤信男監訳)[ソフトバンク]

カーネルは GNU C と GNU ツールチェインを使って書かれています。カーネルは ISO C89 仕様に準拠して書く一方で、標準には無い言語拡張を多く使っています。カーネルは標準 C ライブラリに依存しない、C 言語非依存環境です。そのため、C の標準の中で使えないものもあります。特に任意の long long の除算や浮動小数点は使えません。カーネルがツールチェインや C 言語拡張に置いている前提がどうなっているのかわかりにくいことが時々あり、また、残念なことに決定的なリファレンスは存在しません。情報を得るには、gcc の info ページ (info gcc) を見てください。

あなたは既存の開発コミュニティと一緒に作業する方法を学ぼうとしていることに思い出してください。そのコミュニティは、コーディング、スタイル、開発手順について高度な標準を持つ、多様な人の集まりです。地理的に分散した大規模なチームに対してもっともうまくいくとわかったことをベースにしながら、これらの標準は長い時間をかけて築かれてきました。これらはきちんと文書化されていますから、事前にこれらの標準について事前にできるだけたくさん学んでください。また皆があなたやあなたの会社のやり方に合わせてくれると思わないでください。

### \* 法的問題

Linux カーネルのソースコードは GPL ライセンスの下でリリースされています。ライセンスの詳細については、ソースツリーのメインディレクトリに存在する、COPYING のファイルを見てください。もしライセンスについてさらに質問があれば、Linux Kernel メーリングリストに質問するのではなく、どうぞ法律家に相談してください。メーリングリストの人達は法律家ではなく、法的問題については彼らの声明はあてにするべきではありません。

GPL に関する共通の質問や回答については、以下を参照してください-

<https://www.gnu.org/licenses/gpl-faq.html>

## \* ドキュメント

Linux カーネルソースツリーは幅広い範囲のドキュメントを含んでおり、それらはカーネルコミュニティと会話する方法を学ぶのに非常に貴重なものです。新しい機能がカーネルに追加される場合、その機能の使い方について説明した新しいドキュメントファイルも追加することを勧めます。カーネルの変更が、カーネルがユーザ空間に公開しているインターフェイスの変更を引き起こす場合、その変更を説明するマニュアルページのパッチや情報をマニュアルページのメンテナ [mtk.manpages@gmail.com](mailto:mtk.manpages@gmail.com) に送り、CC を [linux-api@vger.kernel.org](mailto:linux-api@vger.kernel.org) に送ることを勧めます。

以下はカーネルソースツリーに含まれている読んでおくべきファイルの一覧です-

**README** このファイルは Linux カーネルの簡単な背景とカーネルを設定 (訳注 `configure`) し、生成 (訳注 `build`) するために必要なことは何かが書かれています。カーネルに関して初めての人はここからスタートすると良いでしょう。

**Documentation/process/changes.rst** このファイルはカーネルをうまく生成 (訳注 `build`) し、走らせるのに最小限のレベルで必要な数々のソフトウェアパッケージの一覧を示しています。

**Documentation/process/coding-style.rst** これは Linux カーネルのコーディングスタイルと背景にある理由を記述しています。全ての新しいコードはこのドキュメントにあるガイドラインに従っていることを期待されています。大部分のメンテナはこれらのルールに従っているものだけを受け付け、多くの人は正しいスタイルのコードだけをレビューします。

**Documentation/process/submitting-patches.rst** と **Documentation/process/submitting-~~c~~**  
これらのファイルには、どうやってうまくパッチを作つて投稿するかについて非常に詳しく書かれており、以下を含みます (これだけに限らないけれども)

- Email に含むこと
- Email の形式
- だれに送るか

これらのルールに従えばうまくいくことを保証することではありませんが (すべてのパッチは内容とスタイルについて精査を受けるので)、ルールに従わなければ間違いなくうまくいかないでしょう。

この他にパッチを作る方法についてのよくできた記述は-

“The Perfect Patch” <http://www.ozlabs.org/~akpm/stuff/tpp.txt>

“Linux kernel patch submission format” <https://web.archive.org/web/20180829112450/http://linux.yyz.us/patch-format.html>

**Documentation/process/stable-api-nonsense.rst** このファイルはカーネルの中に不变の API を持たないことにした意識的な決断の背景にある理由について書かれています。以下のようなことを含んでいます-

- サブシステムとの間に層を作ること (コンパチビリティのため?)

- オペレーティングシステム間のドライバの移植性
- カーネルソースツリーの素早い変更を遅らせる（もしくは素早い変更を妨げる）

このドキュメントは Linux 開発の思想を理解するのに非常に重要です。そして、他の OS での開発者が Linux に移る時にとても重要です。

**Documentation/admin-guide/security-bugs.rst** もし Linux カーネルでセキュリティ問題を発見したように思ったら、このドキュメントのステップに従ってカーネル開発者に連絡し、問題解決を支援してください。

**Documentation/process/management-style.rst** このドキュメントは Linux カーネルのメンテナ達がどう行動するか、彼らの手法の背景にある共有されている精神について記述しています。これはカーネル開発の初心者なら（もしくは、単に興味があるだけの人でも）重要です。なぜならこのドキュメントは、カーネルメンテナ達の独特な行動についての多くの誤解や混乱を解消するからです。

**Documentation/process/stable-kernel-rules.rst** このファイルはどのように stable カーネルのリリースが行われるかのルールが記述されています。そしてこれらのリリースの中のどこかで変更を取り入れてもらいたい場合に何をすれば良いかが示されています。

**Documentation/process/kernel-docs.rst** カーネル開発に付随する外部ドキュメントのリストです。もしあなたが探しているものがカーネル内のドキュメントでみつからなかった場合、このリストをあたってみてください。

**Documentation/process/applying-patches.rst** パッチとはなにか、パッチをどうやって様々なカーネルの開発ブランチに適用するのかについて正確に記述した良い入門書です。

カーネルはソースコードそのものや、このファイルのようなリストラクチャードテキストマークアップ (ReST) から自動的に生成可能な多数のドキュメントをもっています。これにはカーネル内 API の完全な記述や、正しくロックをかけるための規則などが含まれます。

これら全てのドキュメントを PDF や HTML で生成するには以下を実行します -

```
make pdfdocs  
make htmldocs
```

それぞれメインカーネルのソースディレクトリから実行します。

ReST マークアップを使ったドキュメントは Documentation/output に生成されます。Latex と ePub 形式で生成するには -

```
make latexdocs  
make epubdocs
```

## \* カーネル開発者になるには

もしあなたが、Linux カーネル開発について何も知らないのならば、KernelNewbies プロジェクトを見るべきです

<https://kernelnewbies.org>

このサイトには役に立つメーリングリストがあり、基本的なカーネル開発に関するほとんどどんな種類の質問もできます（既に回答されているようなことを聞く前にまずはアーカイブを調べてください）。またここには、リアルタイムで質問を聞くことができる IRC チャネルや、Linux カーネルの開発に関して学ぶのに便利なたくさんの役に立つドキュメントがあります。

Web サイトには、コードの構成、サブシステム、現在存在するプロジェクト（ツリーにあるもの無いものの両方）の基本的な管理情報があります。ここには、また、カーネルのコンパイルのやり方やパッチの當て方などの間接的な基本情報も記述されています。

あなたがどこからスタートして良いかわからないが、Linux カーネル開発コミュニティに参加して何かすることをさがしているのであれば、Linux kernel Janitor's プロジェクトにいけば良いでしょう -

<https://kernelnewbies.org/KernelJanitors>

ここはそのようなスタートをするのにうってつけの場所です。ここには、Linux カーネルソースツリーの中に含まれる、きれいにし、修正しなければならない、単純な問題のリストが記述されています。このプロジェクトに関わる開発者と一緒に作業することで、あなたのパッチを Linux カーネルツリーに入れるための基礎を学ぶことができ、そしてもしあなたがまだアイディアを持っていない場合には、次にやる仕事の方向性が見えてくるかもしれません。

もしあなたが、すでにひとまとまりコードを書いていて、カーネルツリーに入れたいと思っていたり、それに関する適切な支援を求める場合、カーネルメンターズプロジェクトはそのような皆さんを助けるためにできました。ここにはメーリングリストがあり、以下から参照できます -

<https://selenic.com/mailman/listinfo/kernel-mentors>

実際に Linux カーネルのコードについて修正を加える前に、どうやってそのコードが動作するのかを理解することが必要です。そのためには、特別なツールの助けを借りてでも、それを直接よく読むことが最良の方法です（ほとんどのトリッキーな部分は十分にコメントしてありますから）。そういうツールで特におすすめなのは、Linux クロスリファレンスプロジェクトです。これは、自己参照方式で、索引がついた web 形式で、ソースコードを参照することができます。この最新の素晴らしいカーネルコードのリポジトリは以下で見つかります -

<https://elixir.bootlin.com/>

### \* 開発プロセス

Linux カーネルの開発プロセスは現在幾つかの異なるメインカーネル「ブランチ」と多数のサブシステム毎のカーネルブランチから構成されます。これらのブランチとは -

- Linus のメインラインツリー
- メジャー番号をまたぐ数本の安定版ツリー
- サブシステム毎のカーネルツリー
- 統合テストのための linux-next カーネルツリー

### メインラインツリー

メインラインツリーは Linus Torvalds によってメンテナンスされ、<https://kernel.org> のリポジトリに存在します。この開発プロセスは以下のとおり -

- 新しいカーネルがリリースされた直後に、2週間の特別期間が設けられ、この期間中に、メンテナ達は Linus に大きな差分を送ることができます。このような差分は通常 linux-next カーネルに数週間含まれてきたパッチです。大きな変更は git(カーネルのソース管理ツール、詳細は <http://git-scm.com> 参照) を使って送るのが好ましいやり方ですが、パッチファイルの形式のまま送るのでも十分です。
- 2週間後、-rc1 カーネルがリリースされ、この後にはカーネル全体の安定性に影響をあたえるような新機能は含まない類のパッチしか取り込むことはできません。新しいドライバ(もしくはファイルシステム)のパッチは -rc1 の後で受け付けられることもあることを覚えておいてください。なぜなら、変更が独立していて、追加されたコードの外の領域に影響を与えない限り、退行のリスクは無いからです。-rc1 がリリースされた後、Linus へパッチを送付するのに git を使うこともできますが、パッチはレビューのために、パブリックなメーリングリストへも同時に送る必要があります。
- 新しい -rc は Linus が、最新の git ツリーがテスト目的であれば十分に安定した状態にあると判断したときにリリースされます。目標は毎週新しい -rc カーネルをリリースすることです。
- このプロセスはカーネルが「準備ができた」と考えられるまで継続します。このプロセスはだいたい 6 週間継続します。

Andrew Morton が Linux-kernel メーリングリストにカーネルリリースについて書いたことをここで言っておくことは価値があります -

「カーネルがいつリリースされるかは誰も知りません。なぜなら、これは現実に認識されたバグの状況によりリリースされるのであり、前もって決められた計画によってリリースされるものではないからです。」

## メジャー番号をまたぐ数本の安定版ツリー

バージョン番号が 3 つの数字に分かれているカーネルは -stable カーネルです。これには最初の 2 つのバージョン番号の数字に対応した、メインラインリリースで見つかったセキュリティ問題や重大な後戻りに対する比較的小さい重要な修正が含まれます。

これは、開発/実験的バージョンのテストに協力することに興味が無く、最新の安定したカーネルを使いたいユーザに推奨するブランチです。

安定版ツリーは "stable" チーム <[stable@vger.kernel.org](mailto:stable@vger.kernel.org)> でメンテされており、必要に応じてリリースされます。通常のリリース期間は 2 週間毎ですが、差し迫った問題がなければもう少し長くなることもあります。セキュリティ関連の問題の場合はこれに対してだいたいの場合、すぐにリリースがされます。

カーネルツリーに入っている、[Documentation/process/stable-kernel-rules.rst](#) ファイルにはどのような種類の変更が -stable ツリーに受け入れ可能か、またリリースプロセスがどう動くかが記述されています。

## サブシステム毎のカーネルツリー

それぞれのカーネルサブシステムのメンテナ達は—そして多くのカーネルサブシステムの開発者達も—各自の最新の開発状況をソースリポジトリに公開しています。そのため、自分とは異なる領域のカーネルで何が起きているかを他の人が見られるようになっています。開発が早く進んでいる領域では、開発者は自身の投稿がどのサブシステムカーネルツリーを元にしているか質問されるので、その投稿とすでに進行中の他の作業との衝突が避けられます。

大部分のこれらのリポジトリは git ツリーです。しかしそ他の SCM や quilt シリーズとして公開されているパッチキューも使われています。これらのサブシステムリポジトリのアドレスは MAINTAINERS ファイルにリストされています。これらの多くは <https://git.kernel.org/> で参照することができます。

提案されたパッチがこのようなサブシステムツリーにコミットされる前に、マーリングリストで事前にレビューにかけられます（以下の対応するセクションを参照）。いくつかのカーネルサブシステムでは、このレビューは patchwork というツールによって追跡されます。Patchwork は web インターフェイスによってパッチ投稿の表示、パッチへのコメント付けや改訂などができる、そしてメンテナはパッチに対して、レビュー中、受付済み、拒否というようなマークをつけることができます。大部分のこれらの patchwork のサイトは <https://patchwork.kernel.org/> でリストされています。

## 統合テストのための linux-next カーネルツリー

サブシステムツリーの更新内容がメインラインツリーにマージされる前に、それらは統合テストされる必要があります。この目的のため、実質的に全サブシステムツリーからほぼ毎日プルされてできる特別なテスト用のリポジトリが存在します-

<https://git.kernel.org/?p=linux/kernel/git/next/linux-next.git>

このやり方によって、linux-next は次のマージ機会でどんなものがメインラインにマージされるか、おおまかな展望を提供します。linux-next の実行テストを行う冒険好きなテスターは大いに歓迎されます。

### \* バグレポート

<https://bugzilla.kernel.org> は Linux カーネル開発者がカーネルのバグを追跡する場所です。ユーザは見つけたバグの全てをこのツールで報告すべきです。どう kernel bugzilla を使うかの詳細は、以下を参照してください -

<https://bugzilla.kernel.org/page.cgi?id=faq.html>

メインカーネルソースディレクトリにあるファイル admin-guide/reporting-bugs.rst はカーネルバグらしいものについてどうレポートするかの良いテンプレートであり、問題の追跡を助けるためにカーネル開発者にとってどんな情報が必要なのかの詳細が書かれています。

### \* バグレポートの管理

あなたのハッキングのスキルを訓練する最高の方法のひとつに、他人がレポートしたバグを修正することがあります。あなたがカーネルをより安定化させることに寄与することだけでなく、あなたは現実の問題を修正することを学び、自分のスキルも強化でき、また他の開発者があなたの存在に気がつきます。バグを修正することは、多くの開発者の中から自分が功績をあげる最善の道です、なぜなら多くの人は他人のバグの修正に時間を浪費することを好まないからです。

すでにレポートされたバグのために仕事をするためにには、<https://bugzilla.kernel.org> に行ってください。もし今後のバグレポートについてアドバイスを受けたいのであれば、bugme-new メーリングリスト (新しいバグレポートだけがここにメールされる) または bugme-janitor メーリングリスト (bugzilla の変更毎にここにメールされる) を購読できます。

<https://lists.linux-foundation.org/mailman/listinfo/bugme-new>

<https://lists.linux-foundation.org/mailman/listinfo/bugme-janitors>

### \* メーリングリスト

上のいくつかのドキュメントで述べていますが、コアカーネル開発者の大部分は Linux kernel メーリングリストに参加しています。このリストの登録/脱退の方法については以下を参照してください -

<http://vger.kernel.org/vger-lists.html#linux-kernel>

このメーリングリストのアーカイブは web 上の多数の場所に存在します。これらのアーカイブを探すにはサーチエンジンを使いましょう。例えば -

<http://dir.gmane.org/gmane.linux.kernel>

リストに投稿する前にすでにその話題がアーカイブに存在するかどうかを検索することを是非やってください。多数の事がすでに詳細に渡って議論されており、アーカイブにのみ記録されています。

大部分のカーネルサブシステムも自分の個別の開発を実施するメーリングリストを持っています。個々のグループがどんなリストを持っているかは、MAINTAINERS ファイルにリストがありますので参照してください。

多くのリストは kernel.org でホストされています。これらの情報は以下にあります -

<http://vger.kernel.org/vger-lists.html>

メーリングリストを使う場合、良い行動習慣に従うようにしましょう。少し安っぽいが、以下の URL は上のリスト（や他のリスト）で会話する場合のシンプルなガイドラインを示しています -

<http://www.albion.com/netiquette/>

もし複数の人があなたのメールに返事をした場合、CC: で受ける人のリストはだいぶ多くなるでしょう。正当な理由がない限り、CC: リストから誰かを削除をしないように、また、メーリングリストのアドレスだけにリプライすることのないようにしましょう。1 つは送信者から、もう 1 つはリストからのよう、メールを 2 回受けすることになってもそれに慣れ、しゃれたメールヘッダーを追加してこの状態を変えようとする。人々はそのようなことは好みません。

今までのメールでのやりとりとその間のあなたの発言はそのまま残し、“John Kernelhacker wrote …” の行をあなたのリプライの先頭行にして、メールの先頭でなく、各引用行の間にあなたの言いたいことを追加するべきです。

もしパッチをメールに付ける場合は、Documentation/process/submitting-patches.rst に提示されているように、それはプレーンな可読テキストにすることを忘れないようにしましょう。カーネル開発者は添付や圧縮したパッチを扱いたがりません。彼らはあなたのパッチの行毎にコメントを入れたいので、そうするしかありません。あなたのメールプログラムが空白やタブを圧縮しないように確認しましょう。最初の良いテストとしては、自分にメールを送ってみて、そのパッチを自分で当ててみることです。もしそれがうまく行かないなら、あなたのメールプログラムを直してもらうか、正しく動くように変えるべきです。

何をおいても、他の購読者に対する敬意を表すことを忘れないでください。

## \* コミュニティと共に働くこと

カーネルコミュニティのゴールは可能なかぎり最高のカーネルを提供することです。あなたがパッチを受け入れてもらうために投稿した場合、それは、技術的メリットだけがレビューされます。その際、あなたは何を予想すべきでしょうか？

- 批判
- コメント
- 変更の要求
- パッチの正当性の証明要求
- 沈黙

思い出してください、これはあなたのパッチをカーネルに入れる話です。あなたは、あなたのパッチに対する批判とコメントを受け入れるべきで、それらを技術的レベルで評価して、パッチを再作成するか、なぜこれらの変更をすべきでないかを明確で簡潔な理由の説明を提供してください。もし、あなたのパッチに何も反応がない場合、たまにはメールの山に埋もれて見逃され、あなたの投稿が忘れられてしまうこともあるので、数日待って再度投稿してください。

あなたがやるべきでないことは？

- 質問なしにあなたのパッチが受け入れられると想像すること
- 守りに入ること
- コメントを無視すること
- 要求された変更を何もしないでパッチを出し直すこと

可能な限り最高の技術的解決を求めているコミュニティでは、パッチがどのくらい有益なのかについては常に異なる意見があります。あなたは協調的であるべきですし、また、あなたのアイディアをカーネルに対してうまく合わせるようにすることが望まれています。もしくは、最低限あなたのアイディアがそれだけの価値があるとすんで証明するようにしなければなりません。正しい解決に向かって進もうという意志がある限り、間違うことがあっても許容されることを忘れないでください。

あなたの最初のパッチに単に 1 ダースもの修正を求めるリストの返答になることも普通のことです。これはあなたのパッチが受け入れられないということではありません、そしてあなた自身に反対することを意味するのでもありません。単に自分のパッチに対して指摘された問題を全て修正して再送すれば良いのです。

### \* カーネルコミュニティと企業組織のちがい

カーネルコミュニティは大部分の伝統的な会社の開発環境とは異ったやり方で動いています。以下は問題を避けるためにできると良いことのリストです。

あなたの提案する変更について言うときのうまい言い方 -

- “これは複数の問題を解決します”
- “これは 2000 行のコードを削除します”
- “以下のパッチは、私が言おうとしていることを説明するものです”
- “私はこれを 5 つの異なるアーキテクチャでテストしたのですが…”
- “以下は一連の小さなパッチ群ですが…”
- “これは典型的なマシンでの性能を向上させます…”

やめた方が良い悪い言い方 -

- “このやり方で AIX/ptx/Solaris ではできたので、できるはずだ…”
- “私はこれを 20 年もの間やってきた、だから…”
- “これは私の会社が金儲けをするために必要だ”
- “これは我々のエンタープライズ向け商品ラインのためである”
- “これは私が自分のアイディアを記述した、1000 ページの設計資料である”
- “私はこれについて、6 ヶ月作業している…”
- “以下は…に関する 5000 行のパッチです”

- ・“私は現在のぐちゃぐちゃを全部書き直した、それが以下です…”
- ・“私は〆切がある、そのためこのパッチは今すぐ適用される必要がある”

カーネルコミュニティが大部分の伝統的なソフトウェアエンジニアリングの労働環境と異なるもう一つの点は、やりとりに顔を合わせないということです。email と irc を第一のコミュニケーションの形とする一つの利点は、性別や民族の差別がないことです。Linux カーネルの職場環境は女性や少数民族を受容します。なぜなら、email アドレスによってのみあなたが認識されるからです。国際的な側面からも活動領域を均等にするようにします。なぜならば、あなたは人の名前で性別を想像できないからです。ある男性がアンドレアという名前で、女性の名前はパットかもしれません（訳注 Andrea は米国では女性、それ以外（欧州など）では男性名として使われることが多い。同様に、Pat は Patricia (主に女性名) や Patrick (主に男性名) の略称）。Linux カーネルの活動をして、意見を表明したことがある大部分の女性は、前向きな経験をもっています。

言葉の壁は英語が得意でない一部の人には問題になります。メーリングリストの中で、きちんとアイディアを交換するには、相当うまく英語を操れる必要があることもあります。そのため、自分のメールを送る前に英語で意味が通じているかをチェックすることをお薦めします。

## \* 変更を分割する

Linux カーネルコミュニティは、一度に大量のコードの塊を喜んで受容することはありません。変更是正確に説明される必要があり、議論され、小さい、個別の部分に分割する必要があります。これはこれまで多くの会社がやり慣れてきたことと全く正反対のことです。あなたのプロポーザルは、開発プロセスのとても早い段階から紹介されるべきです。そうすればあなたは自分のやっていることにフィードバックを得られます。これは、コミュニティからみれば、あなたが彼らと一緒にやっているように感じられ、単にあなたの提案する機能のゴミ捨て場として使っているのではない、と感じられるでしょう。しかし、一度に 50 もの email をメーリングリストに送りつけるようなことはやってはいけません、あなたのパッチ群はいつもどんな時でもそれよりは小さくなければなりません。

パッチを分割する理由は以下 -

1) 小さいパッチはあなたのパッチが適用される見込みを大きくします、カーネルの人達はパッチが正しいかどうかを確認する時間や労力をかけないからです。5 行のパッチはメンテナがたった 1 秒見るだけで適用できます。しかし、500 行のパッチは、正しいことをレビューするのに数時間かかるかもしれません（時間はパッチのサイズなどにより指數関数に比例してかかります）

小さいパッチは何かあったときにデバッグもとても簡単になります。パッチを 1 個 1 個取り除くのは、とても大きなパッチを当てた後に（かつ、何かおかしくなった後で）解剖するのに比べればとても簡単です。

2) 小さいパッチを送るだけでなく、送るまえに、書き直して、シンプルにする（もしくは、単に順番を変えるだけでも）ことも、とても重要です。

以下はカーネル開発者の Al Viro のたとえ話です -

“生徒の数学の宿題を採点する先生のことを考えてみてください、先生は生徒が解に到達するまでの試行錯誤を見たいとは思わないでしょう。先生は簡潔な最高の解を見たいのです。良い生徒

はこれを知っており、そして最終解の前の中間作業を提出することは決してないのです

カーネル開発でもこれは同じです。メンテナ達とレビュア達は、問題を解決する解の背後になる思考プロセスを見たいとは思いません。彼らは単純であざやかな解決方法を見たいのです。”

あざやかな解を説明するのと、コミュニティと共に仕事をし、未解決の仕事を議論することのバランスをキープするのは難しいかもしれません。ですから、開発プロセスの早期段階で改善のためのフィードバックをもらうようにするのも良いですが、変更点を小さい部分に分割して全体ではまだ完成していない仕事を(部分的に)取り込んでもらえるようにすることも良いことです。

また、でき上がっていなものや、”将来直す”ようなパッチを、本流に含めてもらうように送っても、それは受け付けられないことを理解してください。

### \* あなたの変更を正当化する

あなたのパッチを分割すると同時に、なぜその変更を追加しなければならないかを Linux コミュニティに知らせることはとても重要です。新機能は必要性と有用性で正当化されなければなりません。

### \* あなたの変更を説明する

あなたのパッチを送付する場合には、メールの中のテキストで何を言うかについて、特別に注意を払ってください。この情報はパッチの ChangeLog に使われ、いつも皆がみられるように保管されます。これは次のような項目を含め、パッチを完全に記述するべきです -

- なぜ変更が必要か
- パッチ全体の設計アプローチ
- 実装の詳細
- テスト結果

これについて全てがどのようにあるべきかについての詳細は、以下のドキュメントの ChangeLog セクションを見てください -

“The Perfect Patch” <http://www.ozlabs.org/~akpm/stuff/tpp.txt>

これらはどれも、実行することが時にはとても困難です。これらの例を完璧に実施するには数年かかるかもしれません。これは継続的な改善のプロセスであり、多くの忍耐と決意を必要とするものです。でも諦めないで、実現は可能です。多数の人がすでにできていますし、彼らも最初はあなたと同じところからスタートしたのですから。

---

Paolo Ciarrocchi に感謝、彼は彼の書いた “Development Process” (<https://lwn.net/Articles/94386/>) セクションをこのテキストの原型にすることを許可してくれました。Randy Dunlap と Gerrit Huizenga はメーリングリストでやるべきこととやってはいけないことのリストを提供してくれました。以下の人々のレビュー、コメント、貢献に感謝。Pat Mochel, Hanna Linder, Randy Dunlap, Kay Sievers, Vojtech Pavlik, Jan Kara, Josh Boyer, Kees Cook, Andrew Morton, Andi Kleen, Vadim Lobanov, Jesper

Juhl, Adrian Bunk, Keri Harris, Frans Pop, David A. Wheeler, Junio Hamano, Michael Kerrisk,  
と Alex Shepard 彼らの支援なしでは、このドキュメントはできなかつたでしょう。

Maintainer: Greg Kroah-Hartman <[greg@kroah.com](mailto:greg@kroah.com)>



---

**CHAPTER  
SIX**

---

**DISCLAIMER**

Translation's purpose is to ease reading and understanding in languages other than English. Its aim is to help people who do not understand English or have doubts about its interpretation. Additionally, some people prefer to read documentation in their native language, but please bear in mind that the *only* official documentation is the English one: `linux_doc`.

It is very unlikely that an update to `linux_doc` will be propagated immediately to all translations. Translations' maintainers - and contributors - follow the evolution of the official documentation and they maintain translations aligned as much as they can. For this reason there is no guarantee that a translation is up to date. If what you read in a translation does not sound right compared to what you read in the code, please inform the translation maintainer and - if you can - check also the English documentation.

A translation is not a fork of the official documentation, therefore translations' users should not find information that differs from the official English documentation. Any content addition, removal or update, must be applied to the English documents first. Afterwards and when possible, the same change should be applied to translations. Translations' maintainers accept only contributions that are merely translation related (e.g. new translations, updates, fixes).

Translations try to be as accurate as possible but it is not possible to map one language directly to all other languages. Each language has its own grammar and culture, so the translation of an English statement may need to be adapted to fit a different language. For this reason, when viewing translations, you may find slight differences that carry the same message but in a different form.

If you need to communicate with the Linux community but you do not feel comfortable writing in English, you can ask the translation's maintainers for help.



## BIBLIOGRAPHY

[it-c-language] <http://www.open-std.org/jtc1/sc22/wg14/www/standards>

[it-gcc] <https://gcc.gnu.org>

[it-clang] <https://clang.llvm.org>

[it-icc] <https://software.intel.com/en-us/c-compilers>

[it-gcc-c-dialect-options] <https://gcc.gnu.org/onlinedocs/gcc/C-Dialect-Options.html>

[it-gnu-extensions] <https://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html>

[it-gcc-attribute-syntax] <https://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html>

[it-n2049] <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2049.pdf>



## INDEX

|  
it\_IT.\_\_futex\_unqueue (*C function*), 1067  
it\_IT.atomic\_dec\_and\_mutex\_lock (*C function*), 1066  
it\_IT.fault\_in\_user\_writeable (*C function*), 1067  
it\_IT.fixup\_pi\_owner (*C function*), 1071  
it\_IT.futex\_exit\_recursive (*C function*), 1068  
it\_IT.futex\_hash (*C function*), 1066  
it\_IT.futex\_lock\_pi\_atomic (*C function*), 1070  
it\_IT.futex\_match (*C function*), 1069  
it\_IT.futex\_proxy\_trylock\_atomic (*C function*), 1072  
it\_IT.futex\_q (*C struct*), 1068  
it\_IT.futex\_queue (*C function*), 1069  
it\_IT.futex\_requeue (*C function*), 1072  
it\_IT.futex\_setup\_timer (*C function*), 1066  
it\_IT.futex\_sleep\_multiple (*C function*), 1075  
it\_IT.futex\_top\_waiter (*C function*), 1067  
it\_IT.futex\_unqueue (*C function*), 1068  
it\_IT.futex\_vector (*C struct*), 1070  
it\_IT.futex\_wait\_multiple (*C function*), 1075  
it\_IT.futex\_wait\_multiple\_setup (*C function*), 1074  
it\_IT.futex\_wait\_queue (*C function*), 1074  
it\_IT.futex\_wait\_requeue\_pi (*C function*), 1073  
it\_IT.futex\_wait\_setup (*C function*), 1075  
it\_IT.get\_futex\_key (*C function*), 1066  
it\_IT.handle\_early\_requeue\_pi\_wakeup (*C function*), 1073  
it\_IT.mutex\_init (*C macro*), 1063  
it\_IT.mutex\_is\_locked (*C function*), 1063  
it\_IT.mutex\_lock (*C function*), 1063  
it\_IT.mutex\_lock\_interruptible (*C function*), 1064  
it\_IT.mutex\_lock\_io (*C function*), 1065  
it\_IT.mutex\_lock\_killable (*C function*), 1065  
it\_IT.mutex\_trylock (*C function*), 1065  
it\_IT.mutex\_unlock (*C function*), 1064  
it\_IT.requeue\_futex (*C function*), 1071  
it\_IT.requeue\_pi\_wake\_futex (*C function*), 1071  
it\_IT.unqueue\_multiple (*C function*), 1074  
it\_IT.wait\_for\_owner\_exiting (*C function*), 1067  
it\_IT.ww\_mutex\_trylock (*C function*), 1064  
it\_IT.ww\_mutex\_unlock (*C function*), 1064