第二部分　进阶篇

第9章　特种文件系统

有一天，身体问心："我要是痛了，医生会给我治，你痛了谁给你治啊？"于是心说："我只能自己给自己治。"也许是因为这样，每个人都有治疗自己心中伤痛的方法。喝酒、唱歌、发火、发疯、找人说话、跑马拉松，等等，等等。当年我上大学时，我们同宿舍的二哥就是喜欢跑马拉松，结果就有一个女孩天天看着他跑，至于接下来怎么样啦，我想你懂的。当然，还有人会逃避这种心里的痛，不过这是我认为最差的方法。我的方法，就是站在讲台上发疯，给大伙讲一些伪技术。一些比较"邪性"的东西。

话说有一种感动，叫内牛满面，有一种文件系统，根本不在磁盘上。这种文件系统就是大名顶顶的 ram-based filesystem。实际上，在 Linux 系统中，/dev、/proc、/sys 目录里面的内容与硬盘是没有半毛钱关系的。那么这些玩意到底有什么用，怎么用，在听我白乎完以后，最好再去看看内核源代码中的一些文档。要知道文档这种东西，真正读起来就嫌少了。至于你信不信，反正我是信了。

在这一章中只是讲这些内容我是不会尽兴的，我还会追加一些譬如tmpfs、debugfs、relayfs等。虽然这似乎是一些很冷的话题，对于很多人来说，它们的受欢迎程度，肯定是既赶不上陈冠希老师的摄影作品，也赶不上苍井空老师的启蒙课程。不过人在江湖身不由己，因为工作的原因，为了提高自己抓紧升P[①]，你不得不来看一看，扩大一下自己的知识面，即便在工作中用不到，以后拿来泡妞或许也能管点用处。有诗云："海上本无花，因风而起；心中本无恨，因爱而生……"

9.1　日志和 ReiserFS

为了追根溯源、舍末逐本，在这一章真正开始之前，我先要脱离一下主题，以便为接下来的行程做好准备。我将会涉及两个对于 Linux 开发社区非常重要的主题——日志和"ReiserFS 后"的设计理念。

日志一直都是非常重要的，就像没有韩局长的日记一样（**请作者解释韩局长的日记是神马东东**），没有它，我们的生活就缺少了很多茶余饭后的谈资。日志一直是人们长期期待的

① 淘宝职位层级有两条线，一条线是 M 序列，一条是 P 序列。技术人员走 P 序列。

技术，从 Linu x 2.4 以后出现了。即使你对日志已有所掌握，我还是希望我有关日志的介绍能够成为一个好的模型，以用来向其他人解释这项技术，或者你就当它是泡妞的工具吧，因为与众不同也是吸引女孩子注意的基本素质。不管你是不是这么认为的，反正我是这么做的，也达到了目的。

本节的后面我会讲解一下 ReiserFS 的设计理念。希望让大家能够认清一个事实，就是新的系统并不是只为了做同样的事情比老的系统快一点，还应该允许我们用以前完全不可能的方法来处理事情。作为一名人民的好干部，如果希望被惦记，可以学我们的郑书记，将自己和蔼可亲的光辉形象搬上台历；作为一名有梦想有追求而又不知道如何出名的人，你可以参考芙蓉和凤姐；作为一个 Linux 爱好者，又想成为 IT 行业中一名能够被人记住的工程师，我们就应该牢记这一点——用以前完全不可能的方法来处理事情。

9.1.1 理解日志

1. 元数据

电影《色戒》告诉我们，床戏是用身体来诠释爱情的；而文件系统则教导我们，元数据就是诠释数据的数据。有点绕，较为通俗的解释是这样的：作为文件系统，一定要提供存储、查询和处理数据的功能。那么，文件系统就保存了一个内部数据结构，使得这些操作成为可能。这个内部数据结构，就是元数据，它为文件系统提供特定的身份和性能特征。

元数据对于 99.99%的人来说，都是不必关心的，因为元数据是专门交给文件系统的驱动程序使用的，平时根本碍不着你什么事儿。不过有一点很重要：要想文件系统的驱动程序好好干活，它就得轻松愉快地找到元数据。要求有三：一要合理、二要一致、三要无干扰。否则的话，驱动程序就没法理解元数据，也操作不了，那么你就只能跟你的文件说拜拜了。

这里啰嗦几句。文件系统是文件系统，文件系统的驱动程序是文件系统的驱动程序，不是一码事，就好比人是人他妈生的，妖是妖他妈生的。那些大家耳熟能详的 ext2、ext3 等，实际上叫文件系统类型。可以这样理解：文件系统是项目，类型是方案，驱动程序就是执行人。虽然不能混淆是非，但是平时跟大家说文件系统，上述三点一般都代表了，不必太较真儿。

2. fsck

既然文件系统驱动程序那么娇贵，就得有人伺候它，给它请个保姆。这个保姆就是 fsck。fsck 确保文件系统驱动程序要用的元数据是干净的，但是有时候也不会特别周到。

它具体是怎么伺候文件系统驱动的呢？是这样的：每次 Linux 启动，在没有挂接任何文件系统的时候，都会启动 fsck 扫描一下/etc/fstab 文件中列出的所有本地文件系统；每次

Linux 关闭，它要把还在内存中的被称之为页面缓存或磁盘缓存中的数据转送到磁盘，还要保证把已经挂接的文件系统卸载干净。这套流程说简单了就是：fsck 要检查那些即将被挂接的文件系统，之前是被卸载干净了的，然后做出一个合理的假设——所有元数据都是干净的，没问题。

3. fsck 的问题

不过话分两头说，当年陈冠希老师对自己的摄影作品也是细心呵护，珍爱有佳，可是偏偏就有意外发生。对于陈老师来说，这个意外对他本人或许还给自己“增光”不少，好多人羡慕得不得了。可是对于文件系统来说，塞翁一旦失马，马还回得来吗？

一般是这样的：当 Linux 遇到异常关机（比如断电、kernel panic 或者管理员有点蛋疼），重启后 fsck 就会发现有文件系统没卸载干净，对应的元数据可能不干净，已经出了问题。于是乎开始奋力苦干，全面审查元数据，修正一切可以修复的错误，文件系统又可以正常使用了。从这点来看，似乎“马”是回来了。

星星还是那颗星星，月亮还是那个月亮，可是“马”还是那匹“马”吗？前面说过，fsck 是修正一切可以修复的错误，那么不能修正的怎么办呢？丢掉——这似乎很残忍，但是也只能丢掉，否则就会像手臂上化了脓的伤口，如果不切掉那块肉，以后失去的可能是整个手臂。其实还有远比这个要严重的，fsck 要扫描全部元数据。这显然不是技巧活儿，是需要动蛮力的。少则花几分钟，多则几小时。如果这事儿发生在任务繁重的数据中心，标准的 fsck 过程就不是在帮你了，那是害你不死啊！

4. 日志

土豆会有的，面包会有的，牛奶也会有的，好运更是会有的。日志——一个更好的解决方案很快就有了。

韩局长的日记里记录了他对女下属都干了些什么，文件系统的日志记录了它对元数据都干了些什么。不同的是，韩局长的日记不单写在纸上，还放到网络里给我们品味人生，文件系统的日志只是放在自己磁盘的分区中，让 fsck 放行的。

具体操作是这样的：元数据出现问题后，fsck 在遇到有日志的文件系统时要做的事情就是放行，接下来由文件系统驱动负责按照日志里面的记载去恢复元数据。具体怎么恢复其实跟 fsck 的方法类似，该扔的还得扔。不过时间可就快多了，毕竟这是技巧活不是体力活。实际结果是，使用日志文件系统修复上百 GB 的元数据也只是眨一下眼时间。

说到这里有人会问。日志只是解决了修复时间的问题，那数据丢失了怎么办？那我告诉你，只能凉拌了。一些建议是：

- 给你的服务器弄个不间断电源；

- 选择一个公认的稳定的内核版本；
- 不要让蛋疼的管理员碰你的服务器。

9.1.2 ReiserFS——卓越的小文件性能与渺茫的未来

说完日志，我们开始说 ReiserFS。选择 ReiserFS 说事儿不单单是因为它是众多日志型文件系统之一，还有它的设计目的也很特别。它的设计者 Hans Reiser 的想法是：一个最好的文件系统，不单能够管理好用户的文件，还能够适应环境干点别的，比如代替数据库。

1. 小文件性能

那么怎么才能让文件系统能够适应环境呢？ReiserFS 就干了这么一件事儿——关注小文件的性能。因为我们常见的如 ext2、ext3 等文件系统一遇到小文件就傻了，这就迫使开发人员在处理比较零碎的数据时，不得不考虑采用数据库或其他手段来获取他们想要得到的性能指标。如此反复下去，问题就来了，在漫漫的历史长河中，这种“针对问题进行创作”的方式怂恿了代码的膨胀，还弄出了一大堆不知所云的带有特殊目的的 API。肮脏的社会，悲催的现实的本源就是那些不合理的制度。

柿子咱们挑软的捏，就拿 ext2 开刀，看看它是怎样把代码肚子搞大的。ext2 比较擅长的是存储大量大小在 20K 以上的文件，但是你要让它帮你存储 20000 个 50 字节左右的文件，ext2 能不能 hold 住不知道，你肯定是 hold 不住了。不但性能急剧下降，存储效率也是飞流直下三千尺。因为 ext2 的最小存储单元是 1k 或 4k，即便你只存储 1 个字节，也得占用 1k 或 4k。这样换算下来，20000×1k=20M 或 20000×4k=80M，而你实际需要的只有 20000×50=1M，存储效率只有 5%，甚至只有 1.25%。如果你头脑还清醒的话，直觉就会告诉你不应该在文件系统上存储这么多小文件。这种经历慢慢地就会成为经验，你也会告诉其他人这么干是不行的，应该使用数据库或者自己想办法。不管怎么样，你都要付出很多额外的代码。

那么 ReiserFS 来了，问题就不这样了。ReiserFS 处理小文件的性能好得不得了。好到什么程度呢？ReiserFS 在处理小于 1k 的文件时，比 ext2 快 8 到 15 倍。更妙的是，处理大于 1k 的文件的时候，也不会有什么性能损失。ReiserFS 大多数情况下都要好于 ext2，处理小文件是它闪光的地方。这个时候你的经验就不管用了，不过也轻松多了。放弃数据库，告别讨厌的数据库访问代码，你要做的就是读写文件。当你用过之后会很惊讶地喊出：原来可以这么简单呀。

2. ReiserFS 技术

ReiserFS 的小文件性能如此突出，那么它是怎么练就这种绝世武功的呢？

原来 ReiserFS 采用了一种叫做 B*树的数据结构。这是一种全新的经过特殊优化的树形数据结构。ReiserFS 用它来组织元数据，相当于整个磁盘分区是一个 B*树。

这里说明一下 B*树的概念。一般专业学过计算机应用的，多少都会接触点数据结构这个东西。不管是老师讲的，还是道听途说来的，链表、堆栈、树、图这些大体上都是了解的。作为“树”这种数据结构，书上说得最多的就是二叉树。但凡涉及二叉树，说得最多的就是二分查找，因为效率高嘛（100 万个数据，只要 20 几次比较就能找到所要的数据）。在二分查找领域中，利用“树”来说事儿的普遍有二叉查找树、平衡二叉查找树，乃至 AVL 树和红黑树。这里大红大紫的当推红黑树，因为 C++的 STL（标准模板库）中的 map 容器就是使用的这种数据结构。乃至后来 Java 中的 map，苹果 Objective-C 中的字典也使用这种数据结构。只要你是程序员，就离不开它。因为在基于内存这种介质的二分查找算法中，红黑树是最稳定的。那么基于磁盘介质上的高效查找（注意这里没有说二分）算法呢？答案是 B 树，正如很多人了解到的“B-”树。其实世界上本没有“B-”树这个东西，只是叫的人多了，它便成为了“B-树”。究其原因是某些人翻译得不负责任，大多数外国文献中使用 B-tree 来说明，就翻译成“B-树”了。实际上 B-tree 说的就是 B 树。B 树是针对磁盘或其他存储介质而设计的一种多叉平衡查找树。与红黑树类似，不同的是，B 树节点可以有很多子女，从几个到几千个。但是马上有人就会站起来反驳，这跟红黑树差别也太大了吧，怎么能说类似呢？是这样的，一棵含有 n 个节点的 B 树的高度也跟红黑树一样，也是 O(logn)，所以 B 树可以在 O(logn)时间内，实现插入、删除等动态集合操作。不过由于分支因子比较大，实际的 B 树要比红黑树的高度小很多。不过实际的文件系统并不是用 B 树，它们大多使用的是 B+树。B+树是 B 树的一个变形，在降低磁盘读写代价的同时，提高了查询效率的稳定性。绕了这么久，终于开始说一下 B*树了，不过你可能会很失望，就是简单一句：B*树是 B+树的变形，B*树分配新节点的概率比 B+树要低，空间利用率更高。

利用 B*树的特性，ReiserFS 允许一个目录下可以容纳 10 万个子目录，从这一点看，就已经基本上排除了文件系统设计上的人为约束了。另外一个好处是，ReiserFS 可以根据需要动态的分配索引，这样也就省去了固定索引，没有附加空间，提高了存储效率。ReiserFS 的与众不同之处还有，不使用固定大小的数据块分配存储空间，采用精确分配原则，这样就不会有磁盘空间的浪费。而且 ReiserFS 还提供了一种叫以尾文件为中心的特殊优化。什么是尾文件呢？就是比系统文件块小的文件或文件的结尾部分。为了提高性能，ReiserFS 可以利用 B*树的叶节点存储文件，从而不用把数据先保存在其他地方，然后再指向它。

ReiserFS 实际上做了两件事。第一，显著提高小文件性能，把文件数据和索引信息放在一起，大多数情况下只需要一次磁盘 I/O 就能搞定；第二，压缩尾文件，这样可以节省大量磁盘空间，一般可以比 ext2 文件系统多存储 6 个百分去点的数据。别小看这 6%，当磁盘足够大，文件足够多时，这方面的性能完全可以用叹为观止来形容。

不过我一向不喜欢把话说死，其实 ReiserFS 的尾文件压缩是以牺牲速度为代价的。有鉴

于这个原因，ReiserFS 的作者们提供了一个开关，可以让管理员关掉尾文件压缩功能，可以让管理员根据实际使用情况，酌情考虑是要速度还是存储能力。

3. 渺茫的未来

对于大多数男人是只知道女人的底裤在那里，却不知道女人的底线在哪里，总是想挑战女人的极限；对于我们德高望重的 Hans Reiser 应该取反，因为他的女人一直在挑战他的极限，结果……耗子急了咬了猫。2008 年 4 月 28 日，被加利福尼亚州奥克兰法庭认定其杀妻罪名成立，判决 15 年监禁。从此 ReiserFS 的开发就基本处于停滞状态。虽然有开发者主动挺身而出，但是主创灵魂已身陷囹圄，ReiserFS 的命运一直蒙着一层阴影。乃至一度大力推广 ReiserFS 的 Novell 公司都开始反水，在 2006 年 10 月 12 日宣布在未来的 SUSE Linux Enterprise 版本中不再使用 ReiserFS 作为默认文件系统，改用 ext3。即便如此，ReiserFS 依然是 Linux 系统中最优秀的文件系统之一，而且现在依然是可以使用的。

9.1.3 应用实战

到目前为止，Linux 的主线版本已经升级到了 3.8，但是很不幸的是，由于 ReiserFS v4 一直没有被纳入 Linux 的主线，我们不得不通过内核补丁来一尝 ReiserFS v4 的朱唇，到本书截稿之前支持的最高内核版本是 3.7。

要使用 ReiserFS v4 还是有点小麻烦的，具体步骤是这样的：

1. 准备内核源代码。这年头怎么都得是 2.6.xx 的吧？
2. 下载一个对应您内核版本的补丁。reiser4-for-2.6.xx.patch.gz。
3. 下载 ReiserFS v4 的工具包。libaal-1.0.5.tar.gz 和 reiser4progs-1.0.6.tar.gz。
4. 安装 libaal。一个工具库，提供个哈希表，位操作什么的。
5. 安装 reiser4progs。这是用来使用 ReiserFS 分区的工具。包括 debugfs.reiser4、fsck.reiser4、measurefs.reiser4 和 mkfs.reiser4。debugfs.reiser4 用来调试 ReiserFS 的，这个工具利用了我们后面要讲解的 debugfs；fsck.reiser4 用来检测和修复 ReiserFS 磁盘分区的，fsck 我们前面已经介绍过了；measurefs.reiser4 用于度量 ReiserFS 磁盘分区，比如查看磁盘碎片；mkfs.reiser4 就是格式化工具了。
6. 给内核源代码打补丁。一般执行的操作就是：
   gzip -cd ../reiser4-for-2.6.xx.patch.gz | patch -p1
   注意：执行这步操作要确保你在内核源代码的根目录下。
7. 配置、编译和安装内核。
8. 使用新内核重新启动。

重新启动后，你就拥有 ReiserFS 了。接下来要做的就是找一个分区，使用 mkfs.reiser4

进行格式化，使用 mount 命令挂载就好了。

9.1.4　小结

我讲这些并不是想让大家如何深刻的理解日志、ReiserFS 乃至什么 B*树。就好比女人如画，不同的画中有不同的风景，Linux 也是如此，左看右看上看下看，角度不同，风景各异。再一次重复之前说的话：用以前完全不可能的方法来完成事情。日志和 ReiserFS 不正是对这句话很好的诠释吗？

接下来我们开始进入正题，Linux 的特种文件系统！

9.2　进程文件系统 procfs

procfs 之于 Linux 的重要程度就好比眼睛之于心。眼睛是心灵的窗口，直达心底；眼睛是心灵感知世界的大门，洞悉全局；眼睛是人与人之间心灵沟通的桥梁，展现彼此。

procfs 是进程文件系统的缩写。这是一个伪文件系统（启动时动态生成的文件系统），用于用户空间通过内核访问进程信息。但是经过不断的演进，如今 Linux 提供的 procfs 已经不单单用于访问进程信息，还是一个用户空间与内核交换数据修改系统行为的接口。这个文件系统通常被挂接到/proc 目录。

procfs 并不是 Linux 的原创，它源自于 UNIX 世界，现在世上几乎所有类 UNIX 系统都提供。可能是历史太过悠久，如今好多人开始讨厌它，排挤它，发明了如 sysfs 这样的东西想要替代它。由于 FreeBSD 已经放弃了 procfs，它默默的承受着：早就说分手，从未被遗弃的命运。因为 procfs 就像气质非凡的美女，虽然朱颜老去，但内在的神韵一直吸引着我们，无法抗拒。

9.2.1　神秘的 9 号计划

procfs 最早在 UNIX 第 8 版实现，后来又移植到了 SVR4，最后由一个被称为“9 号计划”的项目做了大量改进，使得/proc 成为文件系统真正的一部分。“9 号计划”是贝尔实验室创造的另外一个操作系统。这是一个“高尚”的操作系统，一个“纯粹”的操作系统，一个“有道德”的操作系统，一个“脱离了低级趣味”的操作系统，一个“有益于人民”的操作系统。

很久很久以前，贝尔实验室的一群人创造了至今最为重要的网络操作系统——UNIX。曾经有人说过：即便这是贝尔实验室做出的唯一贡献，那也足以让它名垂千古了！到 20 世纪 80 年代中期，计算的趋势从大的集中式的分时计算向更小的个人机器组成的网络方向转

移。人们早已厌倦了既受管束又超载的分时机器，极其渴望使用一种小巧而又自由的系统，缺点就是有点慢。随着微型计算机越来越快，唯一的缺点也可以无视了，于是这种计算方式一直延续到了现在。UNIX 是一个古老的分时系统，很难适应这种计算方式。即便可以让 UNIX 支持图形和网络功能，但办法有点糟，很难管理。更要命的是，这种集中到分散的转化无法做到无缝过度，因为分时是专政和资源集中化，个人计算是民主和资源分散化，而且是从根本上扩大了管理问题。于是，有一些愤青（包括 Dennis Ritchie 和 Ken Thompson），决心依靠自己的经验，超越 UNIX，编写出最完美的操作系统，这就是他们的“9 号计划”。

这是一个“高尚”的操作系统。它的“高尚”在于“对人无所求，给人的却是极好的东西”。它完整的源代码可以免费地在朗讯公共许可证 1.02 版的授权之下取得，而且被开放源代码促进会认为是开放源代码软件，被自由软件基金会认为是自由软件。

这是一个“纯粹”的操作系统。它的“纯粹”在于这么多年来仍作为一个“概念型”的系统存在。它一开始就作为一个完全的网络操作系统被设计。所以在“9 号计划”背后的概念更多的是和网络而不是单个用户的需要相关，它因此而沦落成一种研究用的工具就一点都不奇怪了。所以有人调侃：“这不过是一个操作系统领域用来产生有趣论文的装置。”

这是一个“有道德”的操作系统。它的“有道德”在于它的代码是从底层写起的，并没有包含任何他人的代码。咋看起来它确实和 UNIX 极为相似，但“9 号计划”并不是 UNIX，也不是它的变种，这是一个完完全全的新操作系统。只是操作界面上受到了 UNIX 的很大影响。二者在底层的工作方式完全不同，“9 号计划”最基本的概念是一切皆文件，此技术在 UNIX 下也有利用，但是远没有发展到“9 号计划”的那种程度。

这是一个“脱离了低级趣味”的操作系统。它的“脱离了低级趣味”在于不提倡包干到底，而是要构建一个分工合作的运算环境。比如：单独使用一台具有极强运算能力的计算机用来为远程终端提供运算服务，即专门的 CPU 服务器；同时另有一台专门的机器用来完成存储所有文件的任务，即专门的文件服务器。这样设计的好处就是能够获得管理上的便利和更高的安全性。今天的“云计算”，跟这种思想极为相似。

这是一个“有益于人民”的操作系统。它的“有益于人民”在于其虽并未像 UNIX 一样热门，但是它的精神一直在指引着后继各种操作系统前进的方向。比如微内核概念，Windows NT 和 Mac OS X 受益颇多；/proc 即我们在讲的 procfs 的前身，使得我们与系统内核通信跟读写文件一样简单；提出“网络通透性”概念是现今所有分布式文件系统所追求的目标；引入 Unicode 编码机制，是目前应用最为广泛的文字编码之一。

虽然“9 号计划”在 2002 年宣告终止了，但至目前为止仍在某些领域或被部分业余爱好者当成研究、开发或者使用的操作系统。它最引人注意的地方在于其本身代表了所有的系统接口，除了特殊的接口外，包含了网络接口、用户接口、文件系统接口等。

9.2.2　/proc 目录

如今的/proc 目录已经变得很复杂很复杂，这也是开始排挤它、讨厌它、找人替代它的一个主要出发点。因为如今的 procfs 已经无法满足 UNIX 的 KISS 文化中简单这一条了。但是由于历史原因，至今也无法找到一个更好的办法来完全替代它。那么既然无法反抗，就只能默默享受吧。

大多数情况下，你在/proc 目录下能够看到的文件差不多就是表 9-1 所列出的这些：

表 9-1　/proc 目录下的文件

名　称	功　能	名　称	功　能
apm	高级电源管理信息	loadavg	负载均衡信息
buddyinfo	Buddy 算法内存分配信息	locks	内核锁
compline	内核的命令行参数	mdstat	磁盘阵列状态
config.gz	当前内核的.config 文件	meminfo	内存信息
cpuinfo	cpu 信息	misc	杂项信息
devices	可以用到的设备（块设备/字符设备）	modules	系统已经加载的模块文本列表
diskstats	磁盘 I/O 统计信息	mounts	已挂接的文件系统列表
dma	使用的 DMA 通道	partitions	磁盘分区信息

（续）

名　称	功　能	名　称	功　能
execdomains	执行区域列表	pci	内核识别的 PCI 设备列表
fb	Frame buffer 信息	self	访问 proc 文件系统的进程信息
filesystems	支持的文件系统	slabinfo	内核缓存信息
Interrupt	中断的使用情况，记录中断产生次数	splash	splash 信息
iomem	I/O 内存映射信息	stat	全面统计状态表
ioports	I/O 端口分配情况	swaps	交换空间使用情况
kcore	内核核心映像，GDB 可以利用它查看当前内核的所有数据结构状态	uptime	系统正常运行时间
key-users	密钥保留服务文件	version	内核版本
kmsg	内核消息	vmstat	虚拟内存统计表
ksyms	内核符号表	zoneinfo	内存管理区信息

除了可能会有这些文件外，/proc 目录下还有好多目录。大多数系统会有表 9-2 所列出的这些目录内容：

表 9-2 /proc 目录下的子目录

名 称	功 能	名 称	功 能
[number]	进程信息	irq	中断请求设置接口
acpi	高级配置与电源接口	net	网络各种状态信息
asound	ALSA 声卡驱动接口	scsi	SCSI 设备信息
bus	系统中已安装的总线信息	sys	内核配置接口
dirver	驱动信息	sysvipc	中断的使用情况,记录中断产生次数
fs	文件系统特别信息	tty	tty 驱动信息
Ide	IDE 设备信息		

这里很重要的是[number]这些目录，每个进程一个目录，目录名就是进程 ID。里面包含了一些文件，这些文件描述着一个进程的方方面面，这是 procfs 最初目的的体现。这些文件都是只读的，你不能修改，仅用于获得系统中进程的运行信息。典型的工具如 top、ps 等，就是依据这些目录中的文件所提供的内容进行工作的。

这里有一个特别的目录就是 sys 目录，它所包含的文件大多是可以写的，通过改写这些文件的内容，可以起到修改内核参数的目的。实际上系统命令 sysctl 就是利用这个目录实现的全部功能。使用 C 语言编程时，系统调用的 sysctl 是这个接口的封装。

9.2.3 procfs 实战

对于 procfs 我说的就这么多，我们从实战中可以体会到更多东西，这部分我选择少说多练。

1. 中断平衡

从前，在乡下的时候，是不用排队的，村里的人都很谦让，而且人本来就不多。后来到了县城，县城不大，走亲戚串门或去逛街不用坐车也不用排队，除了街上的游戏厅人多一点外，别的地方人都是不多的，陪妈妈去菜市场买菜也不用排队。后来，到了北京，发现去食堂吃饭要排队，开学报道要排队，在德胜门坐 345 路回学校更要排队。考试挂科去教务处交重修费要排队，甚至连追求一个女孩子也要排队，每次看见人群排成一条长龙时，才真正意识到自己是龙的传人。

其实所有的排队，都是因为资源有限，为了公平所做的一种妥协。但是还有一种现象：在机场，你已经坐在飞机上了，跟你一起的其他飞机有序地等待着有限的跑道资源，时间一分一秒的过去，你的飞机已经晚点一个多小时了。突然你通过舷窗，看到一架飞机直奔云霄，可是你却发现，那架飞机本来是应该排在你后面的，只是因为上面坐着领导。这种情况对于

我这种很阿 Q 的人来说，会念叨一句：“笨鸟先飞。”因为就是不公平了，你能怎么办呢？特权，这就是特权。其实不管你怎样想，特权其实是完全合理的，尤其在 Linux 的世界，合理的利用特权，可以给你的系统带来意想不到的性能提升。

一般情况下，一台计算机，只有一个 CPU，所有设备为了获得 CPU 的青睐，就通过一种叫中断的机制来骚扰一下 CPU。中断被划分成不同的等级，高级别的中断可以被 CPU 优先照顾，同等级别的中断就按照先后顺序排队处理，这在系统内部被称为中断请求队列。高级别中断相对于低阶别中断就有了一种特权。随着时代的变迁，单一 CPU 的设计遇到了瓶颈，无法再继续提高运算能力，计算机开始朝着多核和多 CPU 方向发展。当前主流的服务器配置都可以达到 4CPU16 核心。在这种情况下，相对于数量没有太多变化的外部设备来讲，CPU 不算是一种稀缺资源，但是如何合理的将来自不同设备的中断请求划分给不同的 CPU 就成了一个新的问题。由此引入了一个新的概念，中断平衡。有了中断平衡，我们就又引入了一个新的特权，中断的 CPU 独享特权。即可以指定某颗具体的 CPU 或 CPU 的某颗核心专门处理某个或某些中断请求。

大多数的主流 Linux 发行版都有一个默认的中断平衡策略。但是这些默认的中断平衡策略并不一定能够满足某个特定系统的性能需求，比如一个有着非常繁重的网络资源请求的系统。默认的策略是，网卡的中断请求在多 CPU 环境下，仅发给 CPU0。在一些特定情况下，会导致 CPU0 的资源占用率达到了 100%，而其他 CPU 资源占用却只有 1%~2%，甚至是 0%。由于 CPU0 也要负责任务调度，那么遇到这种情况下，系统基本上就处于死机状态，无法继续正常工作了。解决的办法就是，让网卡把中断请求发给其他 CPU，不过这也需要网卡配合才行，幸好现在大多数服务器所配备的网卡具备这个能力。

那么该如何操作呢？这里先要引入一个概念——中断的 CPU 亲缘性，即中断与哪些 CPU 亲缘。设置好中断的 CPU 亲缘关系，就可以让中断只发往那些它所亲缘的 CPU。

在进行这个设置之前，我们首先要搞清楚，我们的物理设备，到底使用的是哪个中断，每个中断有一个唯一的编号，我们要找到这个编号。可以通过 procfs 的/proc/interrupt 文件来获得。这个文件中的内容差不多是这样的：

```
           CPU0       CPU1
  0:         34          0   IO-APIC-edge      timer
  1:       3435          0   IO-APIC-edge      i8042
  6:          3          0   IO-APIC-edge      floppy
  8:          0          0   IO-APIC-edge      rtc0
  9:      37614          0   IO-APIC-fasteoi   acpi
 12:      12139          0   IO-APIC-edge      i8042
 14:          0          0   IO-APIC-edge      ata_piix
 15:          0          0   IO-APIC-edge      ata_piix
```

```
16:      30317          0   IO-APIC-fasteoi   ahci
17:       3325          0   IO-APIC-fasteoi   82801BA-ICH2
18:         45          0   IO-APIC-fasteoi   uhci_hcd:usb2
19:          0          0   IO-APIC-fasteoi   ehci_hcd:usb1
21:      93253          0   IO-APIC-fasteoi   prl_vtg
22:         26          0   IO-APIC-fasteoi   prl_tg
23:       3259          0   IO-APIC-fasteoi   eth0
```

文件的第一列就是中断号，第二列和第三列是所对应的 CPU 接收到的该中断的数量，最后一列则代表使用该中断的设备，至于倒数第二列，我们不用关心它。从这个例子中网卡，即 eth0 使用了 23 号中断。而且很明显的是只有 CPU0 接收到了中断，CPU1 没有接到过。

找到了对应的中断号，我们可以开始设置它的 CPU 亲缘性了，具体的是设置 procfs 的/proc/irq/[num]/smp_affinity 文件的内容。这个路径下的[num]就是具体的中断号。smp_affinity 文件非常简单，就一个十六进制数，一个位掩码。特定的位对应特定的 CPU，这样，01 就意味着只有第一个 CPU（即 CPU0）可以处理对应的中断，而 0f(1111)就意味着四个 CPU 都会参与中断处理。这样，具体某个中断，可以根据实际系统需求，划分给不同的 CPU 进行处理。

让专门的 CPU 处理专门的中断可以让系统在某些情况下获得极大的性能提升。这也是现代多处理器系统管理上的一个重要观点。比如上面网卡的例子，可以在很大程度上提高系统的可用性和网络吞吐能力，面对突发情况由于 CPU0 依然没有太多负载，系统任务还可以调度，管理员就有机会登录系统，采取必要的维护措施。

2. 获取绝对路径

作为 Linux 程序员，一定会接触到各种各样的配置文件或者日志文件。而且自己写程序，很多时候也喜欢使用配置文件和输出日志文件。在 Linux 中，或者说类 UNIX 系统中，/etc 目录是专门放置配置文件的地方，/var 目录是专门放置日志文件的地方。但是写这两个目录一般需要 root 权限才行，作为小品级别的程序一般是不会考虑使用/etc 和/var 目录的。因此，大多数人会选择跟程序相同的路径或一个跟程序相关的路径。这就引发了一个问题——如何确定配置文件或日志文件的路径。

方法一，使用相对路径。采用相对路径是比较容易也很容易想到的一种方法。通常情况下程序工作的也很好，调试起来也不会有问题。不过当程序投入使用后，会发现一个问题就是：要想正确执行这个程序，就必须进入这个程序所在的目录才行，否则就会找不到配置文件或者日志文件输出路径不对。想让自己的程序成为一个顺手工具，在任意路径下都能正确执行的希望就此破灭。究其原因是因为 fopen、open 等这些打开文件的函数或系统调用在使用相对路径时，默认的当前路径是程序的执行路径，而不是程序所在的路径。换句话说，你

在什么路径下执行这个程序，那么相对路径就是相对于你当前所在的这个路径的。所以自然就会出问题了。可见，这种方法不是一个好方法。

方法二，绝对路径写死。既然采用相对路径会存在相对于当前执行路径的问题，那么干脆使用绝对路径。如果你准备使用/etc 或/var 目录，并且具有 root 执行权限，这是一个不错的主意。但是你要是选择别的路径，就会出现大麻烦。因为不同的管理员有不同的管理规则。比如有些管理员会考虑将一些用户程序统一保存在远程文件服务器中，通过 NFS 共享给所有其他需要这些程序的系统。这时候写死的绝对路径就会出问题，因为不同的系统所挂接的 NFS 磁盘的位置可能不同，那么你的程序就依然无法获得正确的配置文件路径或日志文件输出路径。发生重名的时候，还可能带来更大的麻烦。可见，这依然不是一个好方法。

方法三，获取相对路径转化为绝对路径。具体思路就是，首先决定配置文件或日志文件相对于程序可执行文件的路径，然后再与获取当前可执行文件的绝对路径相结合，即可得到一个新的绝对路径。这个绝对路径在任何情况下都是正确的，不会产生方法一或方法二的问题。该如何获得当前可执行文件的绝对路径呢？最简单直观的就是利用 procfs。

procfs 有一个/proc/self 文件，这实际上是一个神奇的连接。为什么说它神奇呢？是因为不同进程访问这个连接所指向的目标是不同的。为了说明它的神奇，我们先说明一下/proc/[number]目录。其中的[number]是系统中正在运行的进程的 PID，只要启动一个新的进程，在/proc 目录下就会有一个对应的[number]目录出现。那么访问/proc/self 文件的进程所获得的连接目标就是对应的/proc/[pid]路径，即进程所属的进程描述信息目录。

进程描述信息目录中有一个 exe 文件，即/proc/[pid]/exec 文件，这也是一个连接，这个连接的目标是进程可执行文件的绝对路径。只要通过 readlink 系统调用就可以获得一个连接的目标，那么通过/proc/[pid]/exec 文件，就可以获得指定 pid 进程的可执行文件的绝对路径。

获得了绝对路径之后，需要去掉相应的可执行文件名，然后结合相对路径，最终获得配置文件或日志文件的绝对路径。

需要注意的是，你的程序如果是基于多进程方式，并且是采用 Linux 守护进程模式运行的，那么一定要在主进程或调用 daemon 之前完成上述操作，否则将无法获得这个绝对路径。因为对于一个已经丧失亲生父母的孤儿来说，继父无法提供体贴入微的关怀，它的身心怎么会健全呢？

3. 手工释放内存

我在新浪供职时，负责的产品线是新浪 UC。在 IM 领域最为著名的视频聊天室非 UC 莫属。曾在国内的市场占有率高达 90%，直接 P 掉 QQ 不在话下。虽然近年来情况有些飞流直下，但目前其婚恋交友区可以保证每天至少有一对新人成为合法化夫妻。

新浪很穷，真的很穷。UC 视频聊天室在其鼎盛时期，最高同时在线人数超过 60 万，可是只有区区 200 多台 2003 年就开始服役的服务器。排除非业务服务器，平均下来，每台服

务器要为至少 3 千人提供优良的语音视频服务，每台服务器的网口流量超过 600Mbps。我曾多次建议并申请设备采购，但都被驳回。业务在发展，用户数量在不断地增加，后果就是故障率的显著增加。高发的故障率已经开始严重地影响了业务的持续发展。上头下来命令：如果不降低故障率，统统回家！

顿时倍感压力，呼吸困难。于是请各路大神出手帮忙。八路神仙，各显神通使出浑身解数，但是百思不得其钥匙。终于一位大神语出惊吓了一大帮人：你们的程序有内存泄漏，看，空闲内存已经没多少了。于是所有人都将焦点转移到了内存上面。顿时上下翻飞，齐心协力，发现的确任何一台服务器的空闲内存都所剩无几。这下问题大了，就连上头都派人来慰问我们的工作进展如何，并且放话：只要解决内存泄漏问题，今年给你们一等奖金。听到一等奖金，所有人比打鸡血还要兴奋百倍。那就开始吧。不过兴师动众了几个礼拜后，结论是程序没有内存泄漏，但是内存总是被莫名地占用后不被释放。是谁动了我的蛋糕？好吧，请我这个 Linux 伪专家出马吧。

我这里真有一个办法——手工释放内存。操作方法是这样的：

1. 使用 free 命令查看内存。
2. 执行 sync 命令。
3. 执行 echo 3 > /proc/sys/vm/drop_caches。

好了，这就操作完成了。再次使用 free 命令查看内存，会发现剩余内存已经具有惊人的数量了。于是内存真的没有问题了，一等奖金我们拿到了。

不过这个故事没有完。因为故障率就是没有降下来。用户还在抱怨，新业务无法拓展。不过有一个好的消息是，经过这么一折腾，上头居然批准了新设备采购计划。直到新设备上架之后，故障率才显著降低。不过这还不是这个实战的终结。

其实这是一个伪命题！

在 Linux 中，这种手工释放内存的方式是根本解决不了什么内存泄漏问题的，而且还会严重影响系统性能。这是为什么呢？

我先来解释一下这三个步骤的作用：

第一步，使用 free 命令查看内存，这其实没有什么实际作用，就是做个前后对比；

第二步，执行 sync 命令，是为了确保文件系统的完整性（sync 命令将所有未写的系统缓存写到磁盘中）；

第三步，执行 echo 3 > /proc/sys/vm/drop_caches 就开始释放内存了。

这里说明一下/proc/sys/vm/drop_caches 的作用：当写入 1 时，释放页面缓存；写入 2 时，释放目录文件和 inodes；写入 3 时，释放页面缓存、目录文件和 inodes。可见，整个操作过程就是释放磁盘缓存。

Linux 系统与 Windows 在对待内存的问题上是持不同意见的。Linux 会尽量使用内存来提高效率，free 查看剩余内存小并不是说内存不够用，还应该看 swap 是否被大量使用了。

实际项目的经验告诉我们，如果是因为应用程序有内存泄漏、溢出的问题，从 swap 的使用情况是可以比较快速判断的，查看剩余内存是没有意义的或十分困难的。

/proc/sys/vm/drop_caches 是直到 2.6.16 以后的内核版本才开始提供的，我个人认为是内核开发团队对很多用户对 Linux 内存管理方面的疑问的一个妥协，对于是否需要使用这个接口，或向用户提供这个接口，我是持保留意见的。因为当你告诉一个用户，修改一个系统参数可以“释放内存”，剩余内存就多了，用户会怎么想？难道不会觉得这个操作系统“有问题”吗？

这个实战的目的就是想让大家了解，这个接口虽然有提供，但是不要用，因为真的没用。

9.2.4　小结

在 FreeBSD 的开发者们看来，procfs 已经开始背弃了 UNIX 的 KISS 文化，放弃了 procfs 的实现。不过既然我们是来说 Linux 的事儿的，就必须得说 procfs，因为它真的很有用。至于将来 procfs 会怎样，谁也说不准。

Linux 已经开始引入 devfs 和 sysfs 了，这两个怪物有觊觎 procfs 地位之嫌。其中一个已经未老先衰，另一个风华正茂。我在后面会对它们进行介绍。不过我不想这么连续地跟大家讲述这么多跟系统底层相关的部分，毕竟太过乏味。我们先歇歇脚，体验一下 tmpfs 带来的风驰电掣。待轻松之余，我们再去回味一下“底层”工作者们的那份凄凉吧。

9.3　tmpfs——满足你对“时空”的双重渴望

前几天闲来无事翻微薄，有人写道：“曾经偷情被游街，如今二奶喊干爹；曾经撞人忙救人，如今撞人再杀人；曾经私情偷着干，如今淫乱存 U 盘；曾经献血为扶伤，如今慈善越重洋；曾经相好牵肚肠，如今小三炫富忙；曾经摩托都挺酷，如今地铁都追尾；曾经县长坐皮卡，如今少年开宝马；曾经精英成右派，如今牛逼全二代。”不禁感慨万千，这世道真是变了。

曾经内存比金子都贵，现在已经白菜价了。有时候我们在设计系统时，如果磁盘已经忙不过来了，完全可以让内存帮帮忙。不但不会有什么损失，整体执行效率几乎会有一个数量级的提升。tmpfs 就是让你这么干的一个好帮手。

9.3.1　背景

在开始展开对 tmpfs 的论述之前，先要介绍一下 RamDisk。

RamDisk 是将一部分固定大小的内存当作分区来使用。相对于传统的硬盘文件访问来

说，这可以极大地提高在其上进行的文件访问速度。这是一种非常古老的技术，最早可以追溯到上个世纪 80 年代初，我们耳熟能详的 MS-DOS 在其 2.0 版本就加入了对 RamDisk 的支持。自然，这么先进的 Linux 也是不甘寂寞的，将这种技术直接编译进内核了。

对于 RamDisk 的读写与读写普通磁盘分区，从本质上说是没有任何区别的。不过需要注意的是：RamDisk 中的数据是保存在物理内存中的，即便较为先进的实现可以将不常使用的数据交换出去，但也是要占很大一部分内存空间的。而且一旦系统断电，RamDisk 中的任何数据都会随之灰飞烟灭。

不过内存与硬盘在速度上的较量是不言而喻的，稍微懂点计算机技术的人都清楚这点儿。只要合理地利用 RamDisk 是可以得到很大的速度提升的。比如使用 RamDisk 来做 Web 缓存可以极大地提高页面加载速度。

但是随着需求的不断增加以及人们在不知足中的不断探索中，RamDisk 的缺点越来越明显。典型的就是非常浪费物理内存空间。因为你设定的 RamDisk 有多大，它就要占用多大的物理内存，即便你一个字节都没用。所有 RamDisk 都需要进行格式化，如果你选择了一个错误的文件系统，还会造成一定的内存浪费。虽然内存已经不是粒粒如金，但是怎么也得跟大白菜一个价，而硬盘还远不如大白菜值钱呢。必须得改进，应该更高效地利用内存，不能死扣书本，什么时间换空间，空间换时间，这玩意就快要跟封建迷信没什么区别了。一个好的算法，空间性能和时间性能都很好。

另外，在不断的生产实践中，人们发现，大量的临时文件其实很影响程序的性能。于是开始有人把程序产生的临时文件放入 RamDisk 来提高整体性能。其实还是拿 Web 服务器说，大量的缓存文件就可以看作是一种临时文件。因为临时文件有一个特性就是它是临时的，即便丢了，也无大碍。

鉴于上述的一些需求，终于在 Linux 2.4 内核中，引入了一个全新的文件系统——tmpfs，来满足大家对“时空”双重性能的渴望。

9.3.2 tmpfs 文件系统

tmpfs 类似于 RamDisk，它既可以使用内存，也可以使用交换分区。tmpfs 文件系统使用虚拟内存（我们后面简称 VM）子系统的页面来存储文件，tmpfs 自己不需要知道这些页面是在物理内存中还是在交换分区中，一切由 VM 说了算。所以，tmpfs 跟普通的用户进程差不多，使用的只是某种形式的虚拟内存。

tmpfs 的实现与很多人所理解的完全不同，它跟其他文件系统如：ext3、ext2、ReiserFS 等是完全不一致的，它们在 Linux 中都被称为块设备（即读写大块数据的设备，与之相对应的是字符设备，如键盘、鼠标等）。而 tmpfs 是直接建立在 VM 之上的，你用一个简单的 mount 命令就可以创建 tmpfs 文件系统了，不需要什么格式化。事实上就是十分想要格式化你也做

不到，因为地球上就不存在类似 mkfs.tmpfs 这样的命令。

你或许想知道你刚刚挂接地 tmpfs 文件系统到底有多大。这个问题的答案有点意外：不知道！tmpfs 刚被挂接时只有很小的空间，但是随着文件的复制和创建，tmpfs 文件系统驱动程序会分配更多的 VM，并按照需求动态地增加文件系统的空间。当有文件被删除时，tmpfs 文件系统驱动程序会动态地减少文件系统并释放 VM 资源。循环利用，按需分配。因为毕竟 VM 比磁盘金贵些，还是慎用为妙。

说到速度，虽然它使用的是 VM，但是人家也是内存，所以用快如闪电来形容一点都不为过。典型的 tmpfs 文件系统会完全驻留在物理内存中，读写几乎可以说是不用眨眼睛的。即使用了交换分区，性能仍然是卓越的，只要 VM 比较空闲了，一部分 tmpfs 的文件就会被移动到物理内存中，而且不常用的文件，也会被自动交换出去，腾出更多地方给用户进程。显然 tmpfs 遵循着 VM 子系统的整体调度策略，相对于 RamDisk 拥有更好的整体协调性和灵活性。

不过说到底，tmpfs 还是一个基于内存的文件系统，不要指望这种文件系统会提供什么持久性支持，想想都是错误。因为在这个领域看来，那是没有任何意义的功能。而且人家名字也起的好——tmpfs——就是告诉你，别把这儿当安家立业的世外桃源，这里只是一个驿站，风景虽然恬意，但是想留宿，门都没有。

9.3.3　tmpfs 实战

我一直有一个习惯，就是总不甘于就事论事，一定要把大家往“坑”里带，所以接下来的内容大家要注意，“坑”已挖好，就等你来了。

1.　使用 tmpfs

即便 tmpfs 使用起来可以用轻松加愉快来形容，不过我总是要唐僧几句，讲讲如何使用 tmpfs 的，乃至在实际中要遇到的问题。

要使用 tmpfs 最基本的就是要把它挂接到文件系统的某一个节点。只需要使用下面这个简单的命令：

```
#mount tmpfs /tmp –t tmpfs
```

这个时候/tmp 目录就开始使用 tmpfs 文件系统了。所有使用/tmp 目录作为临时目录的程序都会得到很好的速度提升。简单吧？不过注意，问题很快就会出现。最典型的问题就是用光了 VM，虽然不能直接扔给大家一句：“后果自负”。但是你还真得处理好这个后果。

首先，tmpfs 是根据需要动态增大或减少内存的事实就让人有一个疑惑：如果 tmpfs 文件系统增大到耗尽了所有 VM，结果会是怎样？问题到了这个地步，真的很麻烦。早期内核，比如 2.4.4 以前的内核，直接宕机，只能重启了事。虽然 2.4.6 以后的内核做了调整，不过问

题仍然严重，只是不至于宕机罢了。那么会是什么样一个结果呢？不能再向 tmpfs 中写入数据这个自然不在话下，系统还会变得很慢，让你认为它跟宕机差不多了，因为其他程序分配不到内存了。这个时候你让管理员来给你收拾烂摊子，估计也就是重启了事。好了，有人马上会说，你说的都是 2.4 的内核问题，如今的 2.6 乃至 3.0 就不是这样了。的确，但是也好不到那里去。因为内核有一个内建的最后防线，用来在没有可用内存的时候来释放内存。那么它是根据什么来释放内存呢？答案是谁用的内存多，就干掉谁。你马上就会说，显然 tmpfs 占用内存多啊？不过马上你就会很失望，tmpfs 不能被干掉。因为它是内核的一部分，并不是用户进程，而且也没有什么好办法可以让内核找出是哪个进程占满了 tmpfs 文件系统。所以，内核会错误地攻击它能找到的最大的占用内存的程序，通常就是 X 服务器。如果你碰巧在使用它，你的 X 服务器就会被终止，可是引起问题的真凶并没有得到法办。如果碰巧你还在利用 X 提供的图形功能跟刚刚追到的女朋友聊天约会，此时对方刚刚发来见面地点还没来得及看，你是不是会很抓狂呢？

其实 tmpfs 的设计者们早就想到了这个问题，于是提供了一个参数，让你来设定 tmpfs 的最大占用量。可以使用如下命令：

```
# mount tmpfs /tmp -t tmpfs -o size=64m
```

这个命令告诉内核，/tmp 所挂接的 tmpfs 最多只能使用 64MB 的内存。在实际使用中，这个上限未必够用，或者仍然会导致 VM 被用光。比较好的方法是利用 top 工具，来监控一下你的系统在高峰期的内存用量。注意，交换分区要一同算在内。那么高峰期的余量就可以考虑成为 tmpfs 的上限值。不过最好这个上限值再稍微小那么一点，这样可以给你的系统留出一些余量，来应对一下突发事件。

除了容量限制，还可以通过使用 nr_inodes=x 参数限制一下索引节点数量，可以理解为限制了最大的文件数量。这个 x 可以是一个简单的整数，后面还可以跟一个 k、m 或 g 来制定千、百万或十亿个索引节点。

2. 绑定挂接

前面说过，我喜欢把人往“坑”里带，那么现在这个“坑”就在这里了，至于你跟不跟我来，你说了不算！

为什么我说这是一个“坑”呢？因为绑定挂接跟 tmpfs 没有半毛钱关系，引入这个话题其实是为了能够更加灵活地运用 tmpfs，而且通过这个例子，大家可以举一反三，在其他文件系统上同样可以运用绑定挂接这个技术，至少会给自己带来一些方便。有句广告语不是说过嘛：他好，我也好！

那么什么是绑定挂接呢？我无法用一句话清晰明了地概括出来，不过我可以用一个它的行为来描述一下。就是可以通过 mount 命令的一个参数，将一个已经挂接的文件系统全部或

部分挂接到另外一个挂接点上。这里有一个特性（注意，开始挖坑了），任何挂接在绑定挂接文件系统内部的挂接点的文件系统都不会随之挂接。是不是很绕口（显然“坑”很深）？那么我举一个例子说明一下。

我自己的目录是/home/jagen，使用命令“mount --bind / /home/jagen”将系统根目录绑定挂接到了我自己的目录。现在访问我自己的目录跟访问根目录没有任何区别。注意，我自己的目录的访问权限已经跟根目录是相同的了，千万不要自作多情的拿权限开涮，这是没有任何意义的。一般对于根目录的划分是/boot 采用一个分区，/usr 采用一个分区，/home 采用一个分区，/var 采用一个分区，其他的采用一个分区。如果是这么分配的磁盘分区，那么在进行绑定挂接后，我的目录中/home/jagen/usr 这个目录就是空的，其他类推。只有分配给“/”根的这个分区内容被真正挂接到了我的目录（这回应该能从“坑”里爬出来了吧？）。

绑定挂接 tmpfs 的实例是这样操作的，见下面命令：

```
#mkdir /dev/shm/tmp
# chmod 1777 /dev/shm/tmp
# mount --bind /dev/shm/tmp /tmp
```

这里解释一下，/dev/shm 目录就是大多数发行版提供的一个默认的 tmpfs 文件系统，这是 POSIX 标准所规定，因为 POSIX 标准的共享内存就是利用 tmpfs 所实现的。不过目前大家常用的还是 System V 的共享内存，POSIX 的共享内存不是很流行。这只是当前的情况，将来会怎么样，我是说不清楚。既然/dev/shm 就是现成的 tmpfs，那么就在它下面创建一个新的 tmp 目录。注意要修改权限使得所有用户都能访问，因为这是业界针对/tmp 的强制规范。最后使用绑定挂接，将/dev/shm/tmp 这个 tmpfs 绑定挂接到/tmp 上，这样所有使用/tmp 目录作为临时目录的程序都会受益于 tmpfs 所提供的超高性能。另外，这样操作有一个好处就是，/dev/shm 是由发布版本厂商所提供的标准 tmpfs，它的最大容量限制一般可以被认为是最为优秀的，直接拿来用总比自己动手分析要容易得多。

再针对绑定挂接多说几句。绑定挂接对于程序员来说，是非常实用的一个小帮手。我们假设这样一个场景。在一些特定开发场景，为了测试一些新功能，必须修改某个系统文件。但是这个系统文件又是放在只读文件系统上（只读只是相对的，只是修改这个文件非常麻烦罢了），或者这个文件虽然可写，但是对自己没什么把握，不敢直接修改。那么就可以利用 mount --bind 绑定挂接一个新的文件系统，你所有的修改都只是操作这个新的文件系统，老的是不会被改动的。当操作完毕，umount 一下，就完全恢复了。即便弄出问题，重新启动一下，就没有任何问题了。

3. 应用加速

淘宝，作为一个业内最著名的互联网公司，Web 页面是我们对外提供服务的标准接口之一。我想在坐的任何一个人，无不关心 Web 页面的显示速度。那么就叨唠几句用 tmpfs 来加

速 Web 页面吧。

说到 Web，Apache 略显老态龙钟，这个时代已经是 nginx 大行其道的时代，那就看看如何加速 nginx 吧。还记得前面说过的，使得/tmp 目录成为 tmpfs 的方法吧。不过在实际使用中，我们可能不太想改变/tmp 的性质，毕竟还有很多其他程序也要用。那么我们就单独创建一个/nginx_tmp 目录好了。至于是不是这样，你自己决定就行，然后在它上面挂接 tmpfs。可以利用/dev/shm，也可以自己设定大小，随你。然后再修改 nginx.conf 文件，添加如下内容：

```
client_body_temp_path /nginx_tmp/client_body_tmp;
prox_temp_path /nginx_tmp/proxy_temp;
fastcgi_temp_path /nginx_tmp/fastcgi_temp;
```

在 Web 领域，另外一个常用工具当推 Squid 了。这个服务所面临的问题是，当访问量过大时，会急剧增加 Linux 系统的负载（Linux 的系统负载值可以简单理解为是系统进程调度队列中处于等待状态的进程数量）。如果利用 tmpfs，可以有效降低系统负载。一个参考值就是从 12 降到了 0.3，很是可观。具体的操作也非常简单，如何挂接 tmpfs 我不再复述，都是一样的。修改 squid.conf，这个文件的路径一般是/etc/squid/squid.conf。修改内容只有一行，如下：

```
cache_dir ufs /squid_tmp 256 16 256
```

这里对一些参数进行一下解释：

- ufs，缓存存储机制，常用的就是 ufs 机制。不过依赖于操作系统的不同，可以选择不同的存储机制，具体可以参考 Squid 的使用手册。
- /squid_tmp，指定的缓存路径，在这里就是一个 tmpfs 文件系统。
- 第一个 256，指定缓存目录的大小，这是 Squid 能使用的缓存上限，可以根据实际情况酌情设定。
- 16 和第二个 256，针对 ufs 机制，Squid 在缓存目录下创建二级目录树。它们分别制定了第一级和第二级目录的数量。默认就是 16 和 256，你可以根据实际情况，酌情设定

前面讲的基本上都是静态页面的加速，虽然有涉及 FastCGIJ，但是并不能说明全部问题。我再列举一个动态页面加速的例子，利用 tmpfs 如何加速 PHP。对于 Java，道理类似。tmpfs 针对 PHP 的优化主要是提高 session 文件的访问效率，这个只要修改一下 php.ini 文件即可。然后 PHP 程序本身所产生的缓存文件或临时文件，直接在程序中修改就好了。

除了加速 Web 之外，还可以加速一些软件，典型的比如 SQLite 和 BDB。这两个可以算是当今比较流行的桌面级数据库。

SQLite 顾名思义，是一种 SQL 的数据库，拥有优化的 SQL 引擎和数据存储引擎，虽然被称为桌面级数据库，但是性能并不逊色于任何企业级 DBM，结合 tmpfs 可以利用它作为临时表，存放一些临时数据很好用。虽然 SQLite 本身支持内存数据库，但是对于一个老且运行可靠的系统来说，不用修改任何代码不是一个更好的主意吗？

至于 BDB，也就是 Berkeley DB，系出名门，现在已经被 Oracle 收编旗下。BDB 是一个典型的 NoSQL 数据库，性能惊人，支持 Key-Value 存储。最典型的应用就是搜索引擎的网络爬虫。结合 tmpfs 使用，会得到更多意想不到的效果。

当然，对于这些应用要注意的是，不能乱用 tmpfs 造成内存紧张，而且一旦关机所有数据都要灰飞烟灭。不过有一些笨办法可以处理在系统正常关机下的数据转存问题，在后续章节中会有详细介绍。另外也可以自行定义一些持久化策略，使得 tmpfs 文件系统中的内容能够被保存下来。因为在服务器系统设计中，有一个很重要的原则就是 I/O 越少，性能越高，合理利用 tmpfs 可以有效降低系统 I/O 次数，因而提高性能。

9.4　devfs 和 sysfs

devfs 和 sysfs 它们来了，真的来了，一前一后来的，来得是那么突然，来得是那么悄无声息。一个脸色苍白，苍白得让人不寒而栗；一个目光深邃，深邃得让人顿觉谦卑。人们一直在谈论着它们，据说先来的已经死了，死得很透彻，是被它的门人杀死的，而且居然是后来者收买了它的门人，后来的现在还在收买其他门派的门人，正在觊觎“武林盟主”的地位。所有的事情就这么潜移默化地变化着，轮替着。一切看似那么平静，平静得已经让很多人开始摩拳擦掌。在这平静之中不知何时又要到来一场可怕的血雨腥风。

故事是这样开始的……

9.4.1　devfs 的由来

Linux，或者说类 UNIX 系统最“酷”的地方是，设备不是简单地隐藏在晦涩的 API 之后，而是真正的与普通文件、目录或符号连接一样，存在于文件系统之上（还记得我们前面说过的 9 号计划嘛？正是发源于此）。因为字符设备和块设备是映射到普通文件系统名称空间的，这样人们就可以通过很简单的文件读写方式与硬件交互。很多时候仅使用标准的 Linux 命令，如 cat 或 dd，就足够了。这些映射设备的文件被合理的组织在了/dev 目录下。

devfs，也叫设备文件系统，它的唯一目的就是提供一个新的，更合理的方式管理那些位于/dev 目录下的所有块设备和字符设备。因为典型的 Linux 系统是以一种不太理想，而且麻烦的方式管理这些特殊文件的。

时至今日，Linux 支持的硬件种类越来越多，也就意味着在/dev 中的文件数量也越来越多，用数以万计来说的确很夸张，但是要说数以千计、数以百计是绝对不过份的。只是这还不是问题的根本，最根本的是这些特殊文件是写死的，而且大多数根本不会映射到系统中，因为再复杂的服务器，撑死也就配备几十个设备。显然是使用 99.9%的努力，只是为了解决 0.1%的问题。况且谁也不敢保证用户以后不添置什么设备，所以这些文件一个都不能动。

不过 devfs 诞生之际，情况没有上面说的那么糟糕。就是因为 devfs 的生辰问题，导致了它日后的结局，我们祖先发明的生辰八字有些时候想想还是蛮有“科学”道理的。devfs 诞生得太早了，它虽然对上面的问题做了一定的处理，但是有些不是很合理，具体我们后面还会说。现在要说的是，它解决了一个更要命的问题。什么问题呢？设备号的问题。传统的 Linux 设备驱动程序，要向系统提供一个文件映射，需要提供一个主设备号，而且这个主设备号必须保证唯一。由于历史原因，早些年内存比黄金还贵，这个主设备号被设计的只有 8 位，显然这是稀缺资源啊，在它面前，黄金都只能汗颜了。既然这样，开发人员自然不能凭空臆造一个主设备号了，只能联系 Linux 内核的开发人员来申请，如果人家正忙着呢，那您就只能等，还不能歇，一歇就麻烦了，因为等待申请的人多了去了。所以，您就甘心地在那儿耗着吧。直到人家看你是个虔诚的主儿，偶发恻隐之心，给您分配了一个“正式”的主设备号，您才算万事大吉收工交差。其实后面的事情远没有这么简单，只是那已经是历史，我就不多叨唠了。至于这种策略的后果是什么，我不说，谁都知道。反正很难想象，早年的 Linux 用户真是有够虔诚，要不然现在还有谁会知道有 Linux 这个玩意儿呢？

不管 devfs 的命运如何，但就仅仅是把这个滥问题给解决了，就可以称之为伟大，何况这只是其中的一个部分呢？

9.4.2 进入 devfs

devfs 是怎么解决这个滥问题的呢？它给驱动开发人员提供了一个叫 devfs_register()的内核 API，这个 API 可以接受一个设备名称作为参数。调用成功后，在/dev 目录下就会出现与设备名相同的文件名。而且 devfs_register 仍然支持主设备号的策略，这样可以保持向下兼容性，降低早期的设备驱动程序移植的复杂性。

一旦所有设备驱动程序启动并向内核注册适当的设备，内核就启动/sbin/init 进程，系统初始化脚本开始执行。在启动过程初期，rc 脚本将 devfs 文件系统安装在/dev 中，这样/dev 中就包含了 devfs 所表达的所有设备映射关系，所有注册的设备依然可以通过/dev 目录进行访问，用户应用程序不用做任何修改。

这种设计的最大优点就是：所有需要的设备映射关系都由内核自动创建，因此也就不用写死设备文件了，那么/dev 目录下就不会充斥着大量的无用的设备文件了。在实际应用中，只要查看一下 devfs，就能够知道这个系统上有什么设备了。

devfs 让一切变得容易了许多。最典型的就是当你编写一个显示实时系统信息的程序时，不用做依次轮询哪些设备是“活跃的”这样费时的工作。因为只要读取/dev 下的所有信息就可以搞定。即便用户只想查看某一个类型设备的信息，比如光驱，根据 devfs 的约定，只需要读取/dev/cdroms 下的所有文件即可。

在实际操作中，比如你想访问一个特定的块设备，还有很多不同的途径。例如：一个服

务器上，只有一个 SCSI 光驱；使用 devfs 后，就可用通过/dev/cdroms/cdrom0 访问；还通过使用/dev/scsi/host0/bus0/target4/lun0/cd 访问它。这两种都映射了同一个设备，你可以选择一个你认为最方便的途径。如果你愿意，还可以使用一种老式的设备名称/dev/sr0 访问光驱，这都是因为有一个非常便捷的叫 devfsd 的小程序在幕后完成的工作。这个程序虽然小，但是功能很多。它负责创建老式的“兼容性”设备映射文件，允许你以很多方式自定义/dev。

9.4.3　sysfs 的由来

sysfs 是后来的，收买了 devfs 的门徒，杀死了 devfs，它用的不是钱和刀，是 udev。还发表声明公开了 devfs 该杀的四大罪状。但是马上就有人不服了：才四大罪状，好多贪官 100 条大罪都犯下了，也没判死刑不是？Linux 是一个崇尚简单的世界，只要有一条能够说明你很麻烦，就有理由杀掉你，况四条大罪呼？那么这四条大罪是什么呢？

第一，不确定的设备映射，有时一个设备映射的设备文件可能不同，例如我的 U 盘，可能对应 sda 也可能对应 sdb。

第二，没有足够的主/辅设备号，当设备过多的时候，这就是一个问题。前面说过，虽然 devfs 已经意识到了将来的设备会很多，但是没处理好，没有给主/辅设备号太多的扩展余地。

第三，dev 目录下文件太多而且不能表示当前系统上的实际设备（这个罪状在我看来是有点牵强的，不过欲加之罪嘛）。

第四，命名不够灵活，不能任意指定。

于是 devfs 死了，sysfs 成为了新的“帮主”。那么 sysfs 究竟是什么来头呢？系出名门，出身高贵啊。

最初，当人们已经开始意识到 procfs 的复杂度之后，就开始想将 procfs 中有关设备的部分独立出来。最开始采用 ramfs（这个可以看作是 RamDisk 和 tmpfs 的中间产品）作为基础，名曰 ddfs，后来发现 driverfs 更为贴切。这些都是在 2.5 版本中内核鼓捣的。按照那个时候的内核版本号的规则，所有第二位为奇数的版本都是实验版，所以 2.5 这个内核版本对于大众来说是不多见的。driverfs 把实际连接到系统上的设备和总线组织成一个分级的文件，和 devfs 相同，用户空间的程序同样可以利用这些信息以实现和内核的交互（这个思路来自 procfs），该系统是当前实际设备树的一个直观反映。到了 2.6 内核，也就是 2.5 的最终成型版本，新设计了一个 kobject 子系统，它就改变了实现策略抛弃了 ramfs，利用 kobject 子系统来建立这些信息。当一个 kobject 被创建的时候，对应的文件和目录也就被创建了，位于/sys 下的相关目录下。因此更名为 sysfs。因为本身就源自于 procfs 的设计思路，因此它所提供的也是用户空间与系统空间交换信息的接口。用户空间工具 udev 就是利用了 sysfs 提供的信息在用户空间实现了与 devfs 完全相同的功能。既然功能相同，而且是在用户空间实现，显然比在内核空间实现的 devfs 要简单得多，安全得多，也会稳定得多。devfs 被杀，也许这

就是它的宿命。

9.4.4 小结

其实拿 sysfs 和 devfs 做比拼已经没有实际的意义了，因为现在显然已经没有任何争端了。只是我想展现给大家的还是开篇的一个主题：新的系统并不是只为了做同样的事情比老的系统快一点，还应该允许我们用以前完全不可能的方法来处理事情。

显然 sysfs 能够实现全部 devfs 的功能，而且是在用户空间完成的。不单单是这样，当一个并不存在的/dev 节点被打开的时候，devfs 会很负责地去加载这个驱动程序，sysfs 却不会做这种傻事。不过也不能说 devfs 傻，应该说它敬业，想要很负责地告诉用户，这个设备不存在，但是它没有好的机制去做，只能用笨方法，让驱动程序实际去监测设备来报告这个结果。sysfs 如果只是做到这些，应该还是不足以收买 devfs 的门徒的，sysfs 还能给内核产生的设备名增加别名，好处就是用户可以用自己喜欢的名字，显然对用户很友好。sysfs 真正地彻底解决了 devfs 遇到的所有问题。

devfs 和 sysfs 的故事讲到这里也该结束了。在如今的 Linux 发行版中，你再也找不到 devfs 的影子了，但是 procfs 和 sysfs 还在。sysfs 如今大红大紫，procfs 的命运如何，还需要你我共同的期待。

9.5 其他特种文件系统

都说知足者常乐，但是往往就有那么一些人，以发现不足为己任，以满足不足为乐趣。于是就有了四大发明、有了飞机大炮、有了 UNIX、有了 Linux、有了你我今天所面对的世界。

9.5.1 RelayFS

我们做技术的，大多数人都是喜欢买书的，而且买过的书大多都不看。还遭到了信奉“书非借不能读”的人嘲弄。但是在工作中就会发现，那些嘲笑我们只买不看的人终于体会到了书到用时方恨少的苦楚，很快就加入到了我们这类人的行列。其实工具也一样。

前面已经介绍过了，procfs 和 sysfs 是用户空间和内核空间交换数据的接口，但是不满足的人们总是觉得它们不够给力。因为从内核向用户空间反馈大量数据时，无论使用 procfs 还是 sysfs 都是很蹩脚的。IT 界的民工们一直追求的就是高效可靠，越快越好。于是有一种叫 RelayFS 的文件系统就诞生了，它是专门用来从内核空间向用户空间反馈大量数据的。不过最新的称呼，已经叫它 relay 了。我还在叫他 RelayFS 是因为我已经习惯了，改起来还需要一点时间。

在用户空间通过 RelayFS 从内核空间获取数据，是通过 mmap 来完成的。经验丰富的 Linux 程序员一般都会了解 mmap 是读取大块数据的利器。这个在 Windows 系统中叫内存映射文件，其实在 Linux 中也可以这么叫它，把它理解为自实现的轻量化 tmpfs 可以，只是 tmpfs 人家是文件系统，mmap 只能针对一个文件罢了，作用机制差不多，都是利用虚拟内存来提高文件访问效率的。本书后面会有更为详细的内容去介绍 mmap 的原理。

RelayFS 有一个特点，用户空间与内核空间采用通道相连，数据就在通道中传递，要注意的是，这个通道是跟 CPU 一一对应的。它这么设计还是从性能方面考虑的。单 CPU 系统模型就很简单，但是到了多 CPU 时就立马变得复杂起来。简单一点说，因为每个 CPU 都有自己独立的 L1 和 L2 高速缓存。CPU 能够读到数据的唯一来源，本质上是自身的高速缓存。如果高速缓存中没有需要的数据，就需要从内存中先复制到高速缓存。这个过程非常耗费 CPU 的时间，而且一直都出现缓存不命中的情况，整体效率将会非常低下。这个缓存对于应用程序，乃至操作系统都是透明的，谁都控制不了。在现行的 SMP 系统中，即对称式多 CPU 系统中，内存对于任何 CPU 都是共享的，任何一个 CPU 都可以访问内存中的任何一个位置。那么如果不能合理组织数据给不同的 CPU 就会出现问题。例如，当 CPU0 和 CPU1 都要访问同一内存的数据；如果都是读是没有问题的。但是当 CPU0 写这块数据时，实际上写的是它高速缓存内的数据，内存中的数据实际上没有变化。这时 CPU1 要读这部分数据会出现什么现象呢？CPU1 中的缓存失效，不命中。内存中的数据失效，需要 CPU0 同步。于是 CPU0 要将刚才写过的数据同步回内存，CPU1 将内存中的新数据装入缓存。在这个过程中两颗 CPU 的工作效率还不如一颗。所以，RelayFS 为了规避这种情况发生，数据通道采用跟 CPU 一一对应的关系。

9.5.2　debugfs

顾名思义，这就是用于调试用的。不过这个只是用于调试内核用的。应用程序员就不要打它的主意了。

debugfs 是基于 relay 技术实现的。因为 relay 可已极快地将大量的内核空间数据反馈给用户空间，效率是远高于传统的 printk 的。所以使用 debugfs 可以获取更多的调试信息，且占用 CPU 资源更少。由于是基于文件系统的，使用起来会更加方便。毕竟它不会让你的屏幕乱糟糟。

最后说一句就是，想学习内核开发的，就开始使用 debugfs 吧，绝对是你的好帮手。

9.6　结束语

有关 Linux 特种文件系统的一些故事到此就算讲完了。类似 devfs 和 sysfs 这样的惊心动

魄的江湖地位争夺战，在 Linux 世界无时无刻不在上演着。其实我不单单是希望大家通过对特种文件的了解而更明细地认识 Linux，更希望的是通过这一个个案例，来展现 Linux 能够玉树常青的不二法则——新的系统并不是只为了做同样的事情比老的系统快一点，还应该允许我们用以前完全不可能的方法来处理事情，去感受 Linux 世界文化的深邃与博大。就像 sysfs 最终干掉了 devfs。在 Linux 界是允许造反的，只要你能，我们就说：造反无罪，造反有理。