**用 inotify 监控 Linux 文件系统事件**

**在 2.6 内核中进行有效且高效的文件系统事件监控**

当需要对 Linux®文件系统进行高效率、细粒度、异步地监控时，可以采用 inotify。可利用它对用户空间进行安全、性能、以及其他方面的监控。（*2010 年 9 月 10 日，对本文的*[*可下载示例代码*](https://www.ibm.com/developerworks/cn/linux/l-inotify/#download)*进行了更新，更正了一个打印错误。— 编辑注。*）

0https://dw1.s81c.com/developerworks/i/v17/dw-cmts-arrow.png [评论](https://www.ibm.com/developerworks/cn/linux/l-inotify/#icomments)

[Ian Shields](https://www.ibm.com/developerworks/cn/linux/l-inotify/#authorN1002A), 高级程序员, IBM

2010 年 11 月 01 日 (最初于 2010 年 4 月 06 日)

* [expand](https://www.ibm.com/developerworks/cn/linux/l-inotify/#toggle)**内容**

**联系 Ian**

Ian 是我们的最受欢迎、最高产的作者之一。查看有关 developerWorks 的 [Ian 所有的文章](http://www.ibm.com/developerworks/cn/views/global/libraryview.jsp?search_by=ian%20shields)。查看 [Ian 的个人信息](https://www.ibm.com/developerworks/mydeveloperworks/profiles/user/ishields)，和 Ian，其他作者，以及 My developerWorks 的其他读者联系。

在 Linux 内核中对 inotify 进行最后集成之前，我要感谢 IBM 公司 Eli Dow，他写了本文较早的版本。特别要提出的是，在 [下载](https://www.ibm.com/developerworks/cn/linux/l-inotify/#download) 部分所提供的示例代码，大部分还是基于 Eli 最初的样例代码。

**inotify 介绍**

从文件管理器到安全工具，文件系统监控对于的许多程序来说都是必不可少的。从 Linux 2.6.13 内核开始，Linux 就推出了 *inotify*，允许监控程序打开一个独立文件描述符，并针对事件集监控一个或者多个文件，例如打开、关闭、移动/重命名、删除、创建或者改变属性。在后期的内核中有了很多增强，因此在依赖这些特性之前，请先检查您的内核版本。

在本文中，您将会学习如何在简单的监控应用程序中使用 inotify 功能。[下载样例代码](https://www.ibm.com/developerworks/cn/linux/l-inotify/#download)，在您的系统上编译，进一步探索。

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-inotify/#ibm-pcon)

**历史简介**

在 inotify 之前有 dnotify。不幸的是，dnotify 有局限性，用户需要更好的产品。和 dnotify 相比 inotify 的优势如下：

* Inotify 使用一个独立的文件描述符，而 dnotify 需要为每个受监控的目录打开一个文件描述符。当您同时监控多个目录时成本会非常高，而且还会遇到每进程文件描述符限制。
* Inotify 所使用的文件描述符可以通过系统调用获得，并且没有相关设备或者文件。而使用 dnotify，文件描述符就固定了目录，妨碍备用设备卸载，这是可移动媒体的一个典型问题。对于 inotify，卸载的文件系统上的监视文件或目录会产生一个事件，而且监视也会自动移除。
* Inotify 能够监视文件或者目录。Dnotify 则只监视目录，因此程序员还必须保持 stat 结构或者一个等效的数据结构，来反映该被监视目录中的文件，然后在一个事件发生时，将其与当前状态进行对比，以此了解当前目录中的条目发生了什么情况。
* 如上所述，inotify 使用文件描述符，允许程序员使用标准 select 或者 poll 函数来监视事件。这允许高效的多路复用 I/O 或者与 Glib 的 mainloop 的集成。相比之下，dnotify 使用信号，这使得程序员觉得比较困难或者不够流畅。在 2.6.25 内核中 inotify 还添加了 Signal-drive I.O 通知功能。

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-inotify/#ibm-pcon)

**用于 inotify 的 API**

Inotify 提供一个简单的 API，使用最小的文件描述符，并且允许细粒度监控。与 inotify 的通信是通过系统调用实现。可用的函数如下所示：

**inotify\_init**

是用于创建一个 inotify 实例的系统调用，并返回一个指向该实例的文件描述符。

**inotify\_init1**

与 inotify\_init 相似，并带有附加标志。如果这些附加标志没有指定，将采用与 inotify\_init 相同的值。

**inotify\_add\_watch**

增加对文件或者目录的监控，并指定需要监控哪些事件。标志用于控制是否将事件添加到已有的监控中，是否只有路径代表一个目录才进行监控，是否要追踪符号链接，是否进行一次性监控，当首次事件出现后就停止监控。

**inotify\_rm\_watch**

从监控列表中移出监控项目。

**read**

读取包含一个或者多个事件信息的缓存。

**close**

关闭文件描述符，并且移除所有在该描述符上的所有监控。当关于某实例的所有文件描述符都关闭时，资源和下层对象都将释放，以供内核再次使用。

因此，典型的监控程序需要进行如下操作：

1. 使用 inotify\_init 打开一个文件描述符
2. 添加一个或者多个监控
3. 等待事件
4. 处理事件，然后返回并等待更多事件
5. 当监控不再活动时，或者接到某个信号之后，关闭文件描述符，清空，然后退出。

在下一部分中，您将看到可以监控的事件，它们如何在简单的程序中运行。最后，您将看到事件监控如何进行。

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-inotify/#ibm-pcon)

**通告**

当您的应用程序读取到一个通告时，事件的顺序也被读取到您提供的缓存中。事件在一个变长结构中被返回，如清单 1 所示。如果数据占满了您的缓存，您可能需要对最后一个条目进行局部事件信息或者局部名处理。

**清单 1. 用于 inotify 的事件结构体**

struct inotify\_event

{

int wd; /\* Watch descriptor. \*/

uint32\_t mask; /\* Watch mask. \*/

uint32\_t cookie; /\* Cookie to synchronize two events. \*/

uint32\_t len; /\* Length (including NULs) of name. \*/

char name \_\_flexarr; /\* Name. \*/

};

请注意，只有当监控对象是一个目录并且事件与目录内部相关项目有关，而与目录本身无关时，才提供 name 字段。如果 IN\_MOVED\_FROM 事件与相应的 IN\_MOVED\_TO 事件都与被监控的项目有关，cookie 就可用于将两者关联起来。事件类型在掩码字段中返回，并伴随着能够被内核设置的标志。例如，如果事件与目录有关，则标志 IN\_ISDIR 将由内核设置。

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-inotify/#ibm-pcon)

**可监控的事件**

有几种事件能够被监控。一些事件，比如 IN\_DELETE\_SELF 只适用于正在被监控的项目，而另一些，比如 IN\_ATTRIB 或者 IN\_OPEN 则只适用于监控过的项目，或者如果该项目是目录，则可以应用到其所包含的目录或文件。

**IN\_ACCESS**

被监控项目或者被监控目录中的条目被访问过。例如，一个打开的文件被读取。

**IN\_MODIFY**

被监控项目或者被监控目录中的条目被修改过。例如，一个打开的文件被修改。

**IN\_ATTRIB**

被监控项目或者被监控目录中条目的元数据被修改过。例如，时间戳或者许可被修改。

**IN\_CLOSE\_WRITE**

一个打开的，等待写入的文件或目录被关闭。

**IN\_CLOSE\_NOWRITE**

一个以只读方式打开的文件或目录被关闭。

**IN\_CLOSE**

一个掩码，可以很便捷地对前面提到的两个关闭事件（IN\_CLOSE\_WRITE | IN\_CLOSE\_NOWRITE）进行逻辑操作。

**IN\_OPEN**

文件或目录被打开。

**IN\_MOVED\_FROM**

被监控项目或者被监控目录中的条目被移出监控区域。该事件还包含一个 cookie 来实现 IN\_MOVED\_FROM 与 IN\_MOVED\_TO 的关联。

**IN\_MOVED\_TO**

文件或目录被移入监控区域。该事件包含一个针对 IN\_MOVED\_FROM 的 cookie。如果文件或目录只是被重命名，将能看到这两个事件，如果它只是被移入或移出非监控区域，将只能看到一个事件。如果移动或重命名一个被监控项目，监控将继续进行。参见下面的 IN\_MOVE-SELF。

**IN\_MOVE**

可以很便捷地对前面提到的两个移动事件（IN\_MOVED\_FROM | IN\_MOVED\_TO）进行逻辑操作的掩码。

**IN\_CREATE**

在被监控目录中创建了子目录或文件。

**IN\_DELETE**

被监控目录中有子目录或文件被删除。

**IN\_DELETE\_SELF**

被监控项目本身被删除。监控终止，并且将收到一个 IN\_IGNORED 事件。

**IN\_MOVE\_SELF**

监控项目本身被移动。

除了事件标志以外，还可以在 inotify 头文件（/usr/include/sys/inotify.h）中找到其他几个标志。例如，如果只想监控第一个事件，可以在增加监控时设置 IN\_ONESHOT 标志。

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-inotify/#ibm-pcon)

**一个简单 inotify 应用程序**

这里的简单应用程序（参见 [下载](https://www.ibm.com/developerworks/cn/linux/l-inotify/#download) 部分）遵循以上的通用逻辑。我们使用一个信号处理程序来监控 ctrl-c（SIGINT）并且重置一个标志（keep\_running）使应用程序了解终止操作。真实的 inotify 调用在 utility 例程当中完成。注意，我们还创建了一个队列，这样能够将事件从 inotify 底层对象中清除，留着稍后处理。在真实的应用程序中，您可能想用一个不同于您处理事件所用的线程（具有更高优先级）来完成这一操作。对于该应用程序，只是为了对一般原理进行举例说明。我们采用了一个简单的事件链表，在其中我们队列中的每个条目都包含原始事件以及指向队列中下一事件指针的空间。

**主程序**

清单 2 中展示了信号处理例程和主例程。在这个简单示例中，在命令行对每个传递进来的文件会目录建立监控，并利用事件掩码 IN\_ALL\_EVENTS 来监控每个对象的所有事件。在真实的应用程序中，您可能只希望跟踪文件与目录的创建或删除事件，因此您可以屏蔽打开、关闭以及属性改变事件。如果您对文件或目录的重命名和移动不感兴趣，您也可以屏蔽各种移动事件。关于更多细节，参见 inotify 帮助信息。

**清单 2. inotify-test.c 的简单主程序**

/\* Signal handler that simply resets a flag to cause termination \*/

void signal\_handler (int signum)

{

keep\_running = 0;

}

int main (int argc, char \*\*argv)

{

/\* This is the file descriptor for the inotify watch \*/

int inotify\_fd;

keep\_running = 1;

/\* Set a ctrl-c signal handler \*/

if (signal (SIGINT, signal\_handler) == SIG\_IGN)

{

/\* Reset to SIG\_IGN (ignore) if that was the prior state \*/

signal (SIGINT, SIG\_IGN);

}

/\* First we open the inotify dev entry \*/

inotify\_fd = open\_inotify\_fd ();

if (inotify\_fd > 0)

{

/\* We will need a place to enqueue inotify events,

this is needed because if you do not read events

fast enough, you will miss them. This queue is

probably too small if you are monitoring something

like a directory with a lot of files and the directory

is deleted.

\*/

queue\_t q;

q = queue\_create (128);

/\* This is the watch descriptor returned for each item we are

watching. A real application might keep these for some use

in the application. This sample only makes sure that none of

the watch descriptors is less than 0.

\*/

int wd;

/\* Watch all events (IN\_ALL\_EVENTS) for the directories and

files passed in as arguments.

Read the article for why you might want to alter this for

more efficient inotify use in your app.

\*/

int index;

wd = 0;

printf("\n");

for (index = 1; (index < argc) && (wd >= 0); index++)

{

wd = watch\_dir (inotify\_fd, argv[index], IN\_ALL\_EVENTS);

}

if (wd > 0)

{

/\* Wait for events and process them until a

termination condition is detected

\*/

process\_inotify\_events (q, inotify\_fd);

}

printf ("\nTerminating\n");

/\* Finish up by closing the fd, destroying the queue,

and returning a proper code

\*/

close\_inotify\_fd (inotify\_fd);

queue\_destroy (q);

}

return 0;

}

**使用 inotify\_init 打开文件描述符**

清单 3 展示了用于创建 inotify 实例以及获取其文件描述符的简单实用工具函数。文件描述符返回给了调用者。如果出现错误，返回值将为负。

**清单 3. 使用 inotify\_init**

/\* Create an inotify instance and open a file descriptor

to access it \*/

int open\_inotify\_fd ()

{

int fd;

watched\_items = 0;

fd = inotify\_init ();

if (fd < 0)

{

perror ("inotify\_init () = ");

}

return fd;

}

**使用 inotify\_add\_watch 添加一个监控**

一旦我们有了用于 inotify 实例的文件描述符之后，就需要增加一个或多个监控。您可以使用掩码来设置想要监控的事件。在本例中，采用掩码 IN\_ALL\_EVENTS，来监控所有可用事件。

**清单 4. 使用 inotify\_add\_watch**

int watch\_dir (int fd, const char \*dirname, unsigned long mask)

{

int wd;

wd = inotify\_add\_watch (fd, dirname, mask);

if (wd < 0)

{

printf ("Cannot add watch for \"%s\" with event mask %lX", dirname,

mask);

fflush (stdout);

perror (" ");

}

else

{

watched\_items++;

printf ("Watching %s WD=%d\n", dirname, wd);

printf ("Watching = %d items\n", watched\_items);

}

return wd;

}

**事件处理循环**

现在我们已经设置了一些监控，接下来就要等待事件。如果还存在监控，并且 keep\_running 标志没有被信号处理程序重置，则循环会一直进行。循环进程等待事件的发生，对有效事件进行排队，并在返回等待状态之前处理队列。在真实应用程序当中，您可能会将事件放入线程队列中，而在另一个线程中处理它们，清单 5 展示了该循环。

**清单 5. 事件处理循环**

int process\_inotify\_events (queue\_t q, int fd)

{

while (keep\_running && (watched\_items > 0))

{

if (event\_check (fd) > 0)

{

int r;

r = read\_events (q, fd);

if (r < 0)

{

break;

}

else

{

handle\_events (q);

}

}

}

return 0;

}

**等待事件**

在我们的示样例应用程序中，循环会不停地进行下去，直至监控事件出现或者收到了中断信号。清单 6 展示了相关代码。

**清单 6. 等待事件或者中断**

int event\_check (int fd)

{

fd\_set rfds;

FD\_ZERO (&rfds);

FD\_SET (fd, &rfds);

/\* Wait until an event happens or we get interrupted

by a signal that we catch \*/

return select (FD\_SETSIZE, &rfds, NULL, NULL, NULL);

}

**读取事件**

当事件发生时，程序会依照缓存区的大小来读取尽量多的事件，然后把这些事件放入队列等待事件处理程序来处理。样例代码不能处理这种情况 — 可用事件超出 16.384 byte 缓存中可存储的事件。要处理这类情况，需要在缓存末端处理部分事件。目前，对名字长度进行限制不成问题，但是优秀的防御式编程会检查名字，来确保不会溢出缓存。

**清单 7. 读取事件并排队**

int read\_events (queue\_t q, int fd)

{

char buffer[16384];

size\_t buffer\_i;

struct inotify\_event \*pevent;

queue\_entry\_t event;

ssize\_t r;

size\_t event\_size, q\_event\_size;

int count = 0;

r = read (fd, buffer, 16384);

if (r <= 0)

return r;

buffer\_i = 0;

while (buffer\_i < r)

{

/\* Parse events and queue them. \*/

pevent = (struct inotify\_event \*) &buffer[buffer\_i];

event\_size = offsetof (struct inotify\_event, name) + pevent->len;

q\_event\_size = offsetof (struct queue\_entry, inot\_ev.name) +

pevent->len;

event = malloc (q\_event\_size);

memmove (&(event->inot\_ev), pevent, event\_size);

queue\_enqueue (event, q);

buffer\_i += event\_size;

count++;

}

printf ("\n%d events queued\n", count);

return count;

}

**处理事件**

最后，我们需要对事件做处理了。对于该应用程序，我们只简单地报告所发生的事件。如果一个名字出现在事件结构中，我们就报告它是一个文件或目录。发生移动时，还会报告与移动或重命名事件相关的 cookie 信息。清单 8 展示了部分代码，包括对一些事件的处理。参见 [下载](https://www.ibm.com/developerworks/cn/linux/l-inotify/#download) 部分可获取完整代码。

**清单 8. 处理事件**

void handle\_event (queue\_entry\_t event)

{

/\* If the event was associated with a filename, we will store it here \*/

char \*cur\_event\_filename = NULL;

char \*cur\_event\_file\_or\_dir = NULL;

/\* This is the watch descriptor the event occurred on \*/

int cur\_event\_wd = event->inot\_ev.wd;

int cur\_event\_cookie = event->inot\_ev.cookie;

unsigned long flags;

if (event->inot\_ev.len)

{

cur\_event\_filename = event->inot\_ev.name;

}

if ( event->inot\_ev.mask & IN\_ISDIR )

{

cur\_event\_file\_or\_dir = "Dir";

}

else

{

cur\_event\_file\_or\_dir = "File";

}

flags = event->inot\_ev.mask &

~(IN\_ALL\_EVENTS | IN\_UNMOUNT | IN\_Q\_OVERFLOW | IN\_IGNORED );

/\* Perform event dependent handler routines \*/

/\* The mask is the magic that tells us what file operation occurred \*/

switch (event->inot\_ev.mask &

(IN\_ALL\_EVENTS | IN\_UNMOUNT | IN\_Q\_OVERFLOW | IN\_IGNORED))

{

/\* File was accessed \*/

case IN\_ACCESS:

printf ("ACCESS: %s \"%s\" on WD #%i\n",

cur\_event\_file\_or\_dir, cur\_event\_filename, cur\_event\_wd);

break;

/\* File was modified \*/

case IN\_MODIFY:

printf ("MODIFY: %s \"%s\" on WD #%i\n",

cur\_event\_file\_or\_dir, cur\_event\_filename, cur\_event\_wd);

break;

/\* File changed attributes \*/

case IN\_ATTRIB:

printf ("ATTRIB: %s \"%s\" on WD #%i\n",

cur\_event\_file\_or\_dir, cur\_event\_filename, cur\_event\_wd);

break;

/\* File open for writing was closed \*/

case IN\_CLOSE\_WRITE:

printf ("CLOSE\_WRITE: %s \"%s\" on WD #%i\n",

cur\_event\_file\_or\_dir, cur\_event\_filename, cur\_event\_wd);

break;

/\* File open read-only was closed \*/

case IN\_CLOSE\_NOWRITE:

printf ("CLOSE\_NOWRITE: %s \"%s\" on WD #%i\n",

cur\_event\_file\_or\_dir, cur\_event\_filename, cur\_event\_wd);

break;

/\* File was opened \*/

case IN\_OPEN:

printf ("OPEN: %s \"%s\" on WD #%i\n",

cur\_event\_file\_or\_dir, cur\_event\_filename, cur\_event\_wd);

break;

/\* File was moved from X \*/

case IN\_MOVED\_FROM:

printf ("MOVED\_FROM: %s \"%s\" on WD #%i. Cookie=%d\n",

cur\_event\_file\_or\_dir, cur\_event\_filename, cur\_event\_wd,

cur\_event\_cookie);

break;

.

. (other cases)

.

/\* Watch was removed explicitly by inotify\_rm\_watch or automatically

because file was deleted, or file system was unmounted. \*/

case IN\_IGNORED:

watched\_items--;

printf ("IGNORED: WD #%d\n", cur\_event\_wd);

printf("Watching = %d items\n",watched\_items);

break;

/\* Some unknown message received \*/

default:

printf ("UNKNOWN EVENT \"%X\" OCCURRED for file \"%s\" on WD #%i\n",

event->inot\_ev.mask, cur\_event\_filename, cur\_event\_wd);

break;

}

/\* If any flags were set other than IN\_ISDIR, report the flags \*/

if (flags & (~IN\_ISDIR))

{

flags = event->inot\_ev.mask;

printf ("Flags=%lX\n", flags);

}

}

这个简单示例用于说明 inotify 如何工作，以及您可以监控什么事件。您的实际需求将决定对哪些事件进行监控以及如何处理这些事件。

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-inotify/#ibm-pcon)

**用法举例**

在本部分中，我们创建一个带有文件的简单双级目录结构，然后运行简单程序来举例说明 inotify 所能监控的一些事件。我们将在终端会话中启动 inotify 示例程序，但是在后台运行（使用 &）因此程序的输出与我们输入的命令会交替出现。您可以在一个终端窗口运行该程序，而在其他一个或多个窗口运行指令。清单 9 展示了简单文件结构和空文件的创建，以及最初启动该示例程序时的输出。

**清单 9. 创建一个样例环境**

ian@attic4:~/inotify-sample$ mkdir -p dir1/dir2

ian@attic4:~/inotify-sample$ touch dir1/dir2/file1

ian@attic4:~/inotify-sample$ ./inotify\_test dir1/ dir1/dir2/ dir1/dir2/file1&

[2] 8733

ian@attic4:~/inotify-sample$

Watching dir1/ WD=1

Watching = 1 items

Watching dir1/dir2/ WD=2

Watching = 2 items

Watching dir1/dir2/file1 WD=3

Watching = 3 items

ian@attic4:~/inotify-sample$

在清单 10 中，展示了来自 dir2 内容清单的输出。第一个事件报告是关于 dir1 的，展示了一些内容，即 dir2 ，在被监控描述符 1 监控的目录当中被打开了。第二个条目是关于监控描述符 2 的，显示出被监控项目（在本例中为 dir2 ）被打开了。如果正在监控目录树中的多个项目，可能会经常遇到这种双重输出。

**清单 10. 列举 dir2 的内容**

ian@attic4:~/inotify-sample$ ls dir1/dir2

file1

4 events queued

OPEN: Dir "dir2" on WD #1

OPEN: Dir "(null)" on WD #2

CLOSE\_NOWRITE: Dir "dir2" on WD #1

CLOSE\_NOWRITE: Dir "(null)" on WD #2

在清单 11 中，我们在 file1 添加了一些文本。请再次注意对文件以及该文件所在目录的双重打开、关闭和修改事件。还请注意所有的事件并非立刻读取。排队例程被调用了3次，每次有两个事件。如果再次运行该程序，并且每次操作相同，您未必会再次遇到这一特别情况。

**清单 11. 在 file1 中添加文本**

ian@attic4:~/inotify-sample$ echo "Some text" >> dir1/dir2/file1

2 events queued

OPEN: File "file1" on WD #2

OPEN: File "(null)" on WD #3

2 events queued

MODIFY: File "file1" on WD #2

MODIFY: File "(null)" on WD #3

2 events queued

CLOSE\_WRITE: File "file1" on WD #2

CLOSE\_WRITE: File "(null)" on WD #3

在清单 12 中，我们改变了 file1 的属性。我们再次得到有关被监控项目以及其所在目录的双重输出。

**清单 12. 改变文件属性**

ian@attic4:~/inotify-sample$ chmod a+w dir1/dir2/file1

2 events queued

ATTRIB: File "file1" on WD #2

ATTRIB: File "(null)" on WD #3

现在将文件 file1 移动到上一级目录 dir1 当中。在清单 13 中显示了输出结果。这次没有双重条目。我们实际上得到了 3 个条目，每个目录一个，文件本身一个。请注意 cookie (569) 允许将 MOVED-FROM 事件与 MOVED\_TO 事件关联起来。

**清单 13. 将 file1 移动到 dir1**

ian@attic4:~/inotify-sample$ mv dir1/dir2/file1 dir1

3 events queued

MOVED\_FROM: File "file1" on WD #2. Cookie=569

MOVED\_TO: File "file1" on WD #1. Cookie=569

MOVE\_SELF: File "(null)" on WD #3

现在创建一个 file1 到 file2 的硬链接。当到 inode 的链接数量该变时，我们会有一个关于 file1 的 ATTRIB 事件，还会有一个关于 file2 的 CREATE 事件。

**清单 14. 创建硬链接**

ian@attic4:~/inotify-sample$ ln dir1/file1 dir1/file2

2 events queued

ATTRIB: File "(null)" on WD #3

CREATE: File "file2" on WD #1

现在将文件 file1 移入当前目录，将其重命名为 file3 。当前目录没有被监控，因此不存在与 MOVED\_FROM 事件相关联的 MOVED\_TO 事件。

**清单 15. 将 file1 移入不受监控的目录当中**

ian@attic4:~/inotify-sample$ mv dir1/file1 ./file3

2 events queued

MOVED\_FROM: File "file1" on WD #1. Cookie=572

MOVE\_SELF: File "(null)" on WD #3

此时，dir2 是空的，因此可以移动它。注意我们得到一个关于监控描述符 2 的 IGNORED 事件，可见现在我们只监控两个项目。

**清单 16. 移除 dir2**

ian@attic4:~/inotify-sample$ rmdir dir1/dir2

3 events queued

DELETE: Dir "dir2" on WD #1

DELETE\_SELF: File "(null)" on WD #2

IGNORED: WD #2

Watching = 2 items

移除文件 file3。注意这次我们没有得到 IGNORED 事件。为什么呢？为什么得到了关于 file 3 的 ATTRIB 事件（就是原来的 dir1/dir2/file1）？

**清单 16. 删除 file3**

ian@attic4:~/inotify-sample$ rm file3

1 events queued

ATTRIB: File "(null)" on WD #3

记住我们创建了 file1 到 file2 的硬链接。清单 17 显示我们还在通过监控描述符 3 来监控 file2，尽管最开始不存在文件 2！

**清单 17. 我们仍在监视 file2！**

ian@attic4:~/inotify-sample$ touch dir1/file2

6 events queued

OPEN: File "file2" on WD #1

OPEN: File "(null)" on WD #3

ATTRIB: File "file2" on WD #1

ATTRIB: File "(null)" on WD #3

CLOSE\_WRITE: File "file2" on WD #1

CLOSE\_WRITE: File "(null)" on WD #3

因此，现在让我们来删除 dir1，并监控事件级联，因为它不再监控任何事情，不得不结束了自己。

**清单 18. 删除 dir1**

ian@attic4:~/inotify-sample$ rm -rf dir1

8 events queued

OPEN: Dir "(null)" on WD #1

ATTRIB: File "(null)" on WD #3

DELETE\_SELF: File "(null)" on WD #3

IGNORED: WD #3

Watching = 1 items

DELETE: File "file2" on WD #1

CLOSE\_NOWRITE: Dir "(null)" on WD #1

DELETE\_SELF: File "(null)" on WD #1

IGNORED: WD #1

Watching = 0 items

Terminating

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-inotify/#ibm-pcon)

**inotify 的可能使用**

您可以将 inotify 用于多种目标。下面列举一些可能的情况：

**性能监控**

您可能想确定应用程序打开最频繁的文件是哪个。如果发现一个小文件被频繁打开与关闭，您可能会考虑采用内存版，或者改变应用程序来采取其他方式共享该数据。

**元信息**

您可能想记录文件的附加信息，例如起始创建时间或者最后改变该文件的用户 id。

**安全**

您可能会因为安全原因，需要对特定文件或目录的所有访问进行监控。

我们的示例代码监控所有事件并进行报告。实际上，您可能想依据您的需要，来查看这些事件的特定子集。您可能想监控不同被监控项目的不同事件。例如，您可能想监控文件的打开与关闭事件，但对于目录只想监控创建与删除事件。在任何可能的时候，您可以监控您所感兴趣的最小事件集。

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-inotify/#ibm-pcon)

**结束语**

应用到性能监控、程序调试、以及自动化等领域时，inotify 是监控 Linux 文件系统的功能强大的、高粒度的机制。利用本文提供的样例代码，您可以开始编写用来实时记录文件系统事件并最小化性能开销的应用程序。