**seq\_file工作机制实例**

2015年10月16日 20:59:20 [koson\_L](https://me.csdn.net/liaokesen168) 阅读数：902更多

个人分类： [Linux内核](https://blog.csdn.net/liaokesen168/article/category/5884043)

分类： LINUX

**内容简介**

本文主要讲述序列文件（seq\_file）接口的内核实现，如何使用它将Linux内核里面常用的数据结构通过文件（主要关注proc文件）导出到用户空间，最后定义了一些宏以便于编程，减少重复代码。在分析序列文件接口实现的过程中，还连带涉及到一些应用陷阱和避免手段。

**序列文件接口**

UNIX的世界里，文件是最普通的概念，所以用文件来作为内核和用户空间传递数据的接口也是再普通不过的事情，并且这样的接口对于shell也是相当友好的，方便管理员通过shell直接管理系统。由于伪文件系统proc文件系统在处理大数据结构（大于一页的数据）方面有比较大的局限性，使得在那种情况下进行编程特别别扭，很容易导致bug，所以序列文件接口被发明出来，它提供了更加友好的接口，以方便程序员。之所以选择序列文件接口这个名字，应该是因为它主要用来导出一条条的记录数据。  
  
为了能给大家一个具体形象的认识，我们首先来看一段用序列文件接口通过proc文件导出内核双向循环链接表的实例代码:

|  |
| --- |
| #include <linux/kernel.h> #include <linux/module.h> #include <linux/mutex.h> #include <linux/proc\_fs.h> #include <linux/seq\_file.h>  static struct mutex lock; static struct list\_head head; struct my\_data {         struct list\_head list;         int value; };  static void add\_one(void) {         struct my\_data \*data;          mutex\_lock(&lock);         data = kmalloc(sizeof(\*data), GFP\_KERNEL);         if (data != NULL)                 list\_add(&data->list, &head);         mutex\_unlock(&lock); }  static ssize\_t \_seq\_write(struct file \*file, const char \_\_user \* buffer,                        size\_t count, loff\_t \*ppos) {         add\_one();         return count; }  static int \_seq\_show(struct seq\_file \*m, void \*p) {         struct my\_data \*data = list\_entry(p, struct my\_data, list);          seq\_printf(m, "value: %d\n", data->value);         return 0; }  static void \*\_seq\_start(struct seq\_file \*m, loff\_t \*pos) {         mutex\_lock(&lock);         return seq\_list\_start(&head, \*pos); }  static void \*\_seq\_next(struct seq\_file \*m, void \*p, loff\_t \*pos) {         return seq\_list\_next(p, &head, pos); }  static void \_seq\_stop(struct seq\_file \*m, void \*p) {         mutex\_unlock(&lock); }  static struct seq\_operations \_seq\_ops = {         .start = \_seq\_start,         .next = \_seq\_next,         .stop = \_seq\_stop,         .show = \_seq\_show };  static int \_seq\_open(struct inode \*inode, struct file \*file) {         return seq\_open(file, &\_seq\_ops); }  static struct file\_operations \_seq\_fops = {         .open = \_seq\_open,         .read = seq\_read,         .write = \_seq\_write,         .llseek = seq\_lseek,         .release = seq\_release };  static void clean\_all(struct list\_head \*head) {         struct my\_data \*data;          while (!list\_empty(head)) {                 data = list\_entry(head->next, struct my\_data, list);                 list\_del(&data->list);                 kfree(data);         } }  static int \_\_init init(void) {         struct proc\_dir\_entry \*entry;          mutex\_init(&lock);         INIT\_LIST\_HEAD(&head);          add\_one();         add\_one();         add\_one();          entry = create\_proc\_entry("my\_data", S\_IWUSR | S\_IRUGO, NULL);         if (entry == NULL) {                 clean\_all(&head);                 return -ENOMEM;         }         entry->proc\_fops = &\_seq\_fops;          return 0; }  static void \_\_exit fini(void) {         remove\_proc\_entry("my\_data", NULL);         clean\_all(&head); }  module\_init(init); module\_exit(fini); |

注：以上代码需要内核版本不小于2.6.23，小版本内核可以从2.6.23内核代码中抽取函数seq\_list\_start和seq\_list\_next。  
  
上面的这段程序创建了一个结构体my\_data的双向循环链表(head)，并且通过my\_data proc文件将其导出到用户空间，用户空间的程序除了可以通过对my\_data的读操作来获取my\_data链表中各个结构体中的value域的值之外，还能通过写(write）系统调用添加一个具有随机value值的my\_data结构体到双向循环链表中（从链表的头部添加）。

|  |
| --- |
| gentux kernel # cat /proc/my\_data  value: 474615376 value: 474615632 value: 474615568 gentux kernel # echo 0 > /proc/my\_data  gentux kernel # cat /proc/my\_data  value: 758528120 value: 474615376 value: 474615632 value: 474615568 |

虽然，上面的那段代码行数不少，但是和输出相关的部分除了函数\_seq\_show外都是惯例代码，这种简便性是很直观的。相信对于双向循环链表数据结构的导出，读者们都能够照猫画虎写出自己的代码而毋须我再多言。  
  
序列文件接口相关的两个重要数据结构为seq\_file结构体和seq\_operations表（表是延续ULK3的叫法，实际上就是只包含函数指针的结构体)。首先介绍seq\_file结构体：

* buf: 序列文件对应的数据缓冲区，要导出的数据都是首先打印到这个缓冲区，然后才被拷贝到用户指定的用户空间缓冲区。
* size: 缓冲区的大小，默认为1个内存页面大小，随着需求会动态以2的级数倍扩张，比如：4k，8k，16k...
* from: 没有拷贝到用户空间的数据在buf中的起始偏移量。
* count: buf中没有被拷贝到用户空间的数据的字节数。
* index: 数据项的索引，和稍后提到的seq\_operations表中的start, next操作中的pos项一致。
* version: 文件的版本。
* lock: 序列化对这个文件的并行操作
* op：指向稍后提到的seq\_operations表。
* private： 指向文件的私有数据，是特例化一个序列文件的方法。

seq\_operations表中的函数指针成员如下:

* start: 开始读数据项，通常需要在这个函数里面加锁，以防止并行访问数据。
* stop: 停止读数据项，和start相对，通常需要解锁。
* next：找到下一个要处理的数据项。
* show：打印数据项到临时缓冲区。

其中start，next返回的值都会以第二个参数的形式传递给stop和show。start在\*pos为0的时候还可以返回SEQ\_START\_TOKEN，通常这个值传递给show的时候，show会打印表格头。  
  
一些有用的全局函数:

* seq\_open：通常会在打开文件的时候调用，以第二个参数为seq\_operations表创建seq\_file结构体。
* seq\_read, seq\_lseek和seq\_release：他们通常都直接对应着文件操作表中的read, llseek和release。
* seq\_escape：将一个字符串中的需要转义的字符（字节长）以8进制的方式打印到seq\_file。
* seq\_putc, seq\_puts, seq\_printf：他们分别和C语言中的putc，puts和printf相对应。
* seq\_path：用于输出文件名。
* single\_open, single\_release: 打开和释放只有一条记录的文件。
* seq\_open\_private, \_\_seq\_open\_private, seq\_release\_private：和seq\_open类似，不过打开seq\_file的时候创建一小块文件私有数据。

在2.6.23中，又为双向循环链表引入了三个帮助函数:

* seq\_list\_start：返回链表中的特定项。
* seq\_list\_start\_head：在需要输出表格头的时候使用。
* seq\_list\_next: 返回链表中的下一项。

这三个函数的具体用法可以参考上面的实例，不再赘述。

**序列文件接口的内核实现**

序列文件接口的实现比较简单易懂，出于完整性考虑，我姑且把核心函数seq\_read拿来和读者一起解读：

1. 如果是第一次对序列文件调用读操作，那么内核数据缓冲区指针（buf）为空，这个时候申请一整页内存作为缓冲区。
2. 如果内核缓冲区中尚且有数据，那么先用它们填补用户缓冲区。
3. 如果用户缓冲区仍有空间，则按序调用start和show，填补内核缓冲区。如果当前的内核缓冲区不足以容纳一条记录，那么将缓冲区大小加倍后再次尝试，直到可以容纳至少一条记录。然后持续调用next和show，直到内核缓冲区无空间容纳新的整条记录后调用stop终止。

可见，序列文件接口是面向整条记录的，但是它只能保证这些，无法保证导出的数据结构的整体一致性，这一点请多加留意，防止因为它导致bug。  
  
因为序列文件的缓冲区有自动扩张的功能，所以它更便于导出大于一页的数据结构，这也正是single\_open/release函数的用武之地。

**内核常用数据结构的导出**

应用序列文件接口导出常用的面向记录的数据结构通常是非常简单的，下面我将给出内核常用的数据结构通过序列文件接口导出的思路。

**数组**

数组通过其索引就能够随机地访问其中的任一元素，所以我们只要将start和next参数中的pos当成索引，返回对应元素的地址即可。

**双向循环链表**

* start：跳过前pos个元素，返回即可。
* next：增加pos，返回下一个元素。

**哈希表**

哈希表有两种导出方法：

1. 以每个哈希桶为元素，这个时候哈希表退化为数组。
2. 以每个哈希节点为元素，这个时候它退化为按桶连接起来的链表。

第1个方法，因为每个桶中的节点数不可控，所以可能需要大于一页的连续内存，这在一个运行了很长时间，内存充斥着外部碎片的系统上可能是不切实际的。相比之下，第2个方法可能更加有效。

**红黑树**

因为Linux内核提供了按照类似链表的方式遍历红黑树的方法，所以导出它的方法和双向链接表类似。

**序列文件接口的帮助宏**

实际上，应用序列文件接口进行编程，存在大量的重复代码，为了减少可恶的重复代码，也为了减少复制粘贴可能引入的bug，我写了这一套宏以简化大多数情况下的序列文件接口调用。具体宏代码如下:

|  |
| --- |
| #ifndef \_SEQ\_FILE\_HELPER\_H\_ #define \_SEQ\_FILE\_HELPER\_H\_  #include <linux/version.h> #include <linux/list.h> #include <linux/rbtree.h> #include <linux/seq\_file.h>  #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2, 6, 23) static inline struct list\_head \*seq\_list\_start(struct list\_head \*head,                 loff\_t pos) {         struct list\_head \*lh;          list\_for\_each(lh, head)                 if (pos-- == 0)                         return lh;          return NULL; }  static inline struct list\_head \*seq\_list\_next(void \*v, struct list\_head \*head,                 loff\_t \*ppos) {         struct list\_head \*lh;          lh = ((struct list\_head \*)v)->next;         ++\*ppos;         return lh == head ? NULL : lh; } #endif  /\* common macro \*/ #define \_\_DEFINE\_FOO\_SEQ\_FOPS(prefix, foo, imp\_show, imp\_write) \ static struct seq\_operations \_##foo##\_seq\_ops = { \         .start = \_##foo##\_seq\_start, \         .next = \_##foo##\_seq\_next, \         .stop = \_##foo##\_seq\_stop, \         .show = imp\_show \ }; \  \ static int \_##foo##\_seq\_open(struct inode \*inode, struct file \*file) \ { \         return seq\_open(file, &\_##foo##\_seq\_ops); \ } \  \ prefix struct file\_operations foo##\_seq\_fops = { \         .open = \_##foo##\_seq\_open, \         .read = seq\_read, \         .write = imp\_write, \         .llseek = seq\_lseek, \         .release = seq\_release, \          .owner = THIS\_MODULE  };  /\* helper for array \*/ #define \_DEFINE\_ARRAY\_SEQ\_FOPS(prefix, arr, n, lock, unlock, imp\_show, \                 imp\_write) \ static void\* \_##arr##\_seq\_start(struct seq\_file \*m, loff\_t \*pos) \ { \         lock; \         if (\*pos >= n) \                 return NULL; \         return &arr[\*pos]; \ } \  \ static void\* \_##arr##\_seq\_next(struct seq\_file \*m, void \*p, loff\_t \*pos) \ { \         if (++\*pos >= n) \                 return NULL; \         return &arr[\*pos]; \ } \ \ static void \_##arr##\_seq\_stop(struct seq\_file \*m, void \*p) \ { \         unlock; \ } \  \ \_\_DEFINE\_FOO\_SEQ\_FOPS(prefix, arr, imp\_show, imp\_write)  #define DEFINE\_ARRAY\_SEQ\_FOPS(arr, n, lock, unlock, imp\_show, imp\_write) \         \_DEFINE\_ARRAY\_SEQ\_FOPS(, arr, n, lock, unlock, imp\_show, imp\_write)  #define DEFINE\_STATIC\_ARRAY\_SEQ\_FOPS(arr, n, lock, unlock, imp\_show, \                 imp\_write) \         \_DEFINE\_ARRAY\_SEQ\_FOPS(static, arr, n, lock, unlock, imp\_show, \                         imp\_write)  /\* helper for list \*/ #define \_DEFINE\_LIST\_SEQ\_FOPS(prefix, list, lock, unlock, imp\_show, imp\_write) \ static void\* \_##list##\_seq\_start(struct seq\_file \*m, loff\_t \*pos) \ { \         lock; \         return seq\_list\_start(&list, \*pos); \ } \  \ static void\* \_##list##\_seq\_next(struct seq\_file \*m, void \*p, loff\_t \*pos) \ { \         return seq\_list\_next(p, &list, pos); \ } \ \ static void \_##list##\_seq\_stop(struct seq\_file \*m, void \*p) \ { \         unlock; \ } \  \ \_\_DEFINE\_FOO\_SEQ\_FOPS(prefix, list, imp\_show, imp\_write)  #define DEFINE\_LIST\_SEQ\_FOPS(list, lock, unlock, imp\_show, imp\_write) \         \_DEFINE\_LIST\_SEQ\_FOPS(, list, lock, unlock, imp\_show, imp\_write)  #define DEFINE\_STATIC\_LIST\_SEQ\_FOPS(list, lock, unlock, imp\_show, imp\_write) \         \_DEFINE\_LIST\_SEQ\_FOPS(static, list, lock, unlock, imp\_show, imp\_write)  /\* helper for hlist \*/ #ifndef hlist\_for\_each\_continue #define hlist\_for\_each\_continue(pos) \         for (pos = pos->next; pos && ({ prefetch(pos->next); 1; }); \                         pos = pos->next) #endif  #define \_DEFINE\_HLIST\_SEQ\_FOPS(prefix, head, n, lock, unlock, imp\_show, \                 imp\_write) \ struct \_##head##\_seq\_iter { \         unsigned int idx; \         struct hlist\_node \*node; \ }; \  \ static void\* \_##head##\_seq\_start(struct seq\_file \*m, loff\_t \*pos) \ { \         struct hlist\_node \*node; \         unsigned int idx; \         loff\_t off = \*pos; \  \         lock; \         for (idx = 0; idx < n; idx++) { \                 hlist\_for\_each(node, &head[idx]) { \                         if (off-- == 0) { \                                 struct \_##head##\_seq\_iter \*iter; \                                 iter = kmalloc(sizeof(\*iter), GFP\_KERNEL); \                                 if (iter == NULL) \                                         break; \                                 iter->idx = idx; \                                 iter->node = node; \                                 return iter; \                         } \                 } \         } \  \         return NULL; \ } \  \ static void\* \_##head##\_seq\_next(struct seq\_file \*m, void \*p, loff\_t \*pos) \ { \         struct \_##head##\_seq\_iter \*iter = (struct \_##head##\_seq\_iter\*)p; \  \         ++\*pos; \         hlist\_for\_each\_continue(iter->node) \                 return iter; \         for (iter->idx++; iter->idx < n; iter->idx++) { \                 hlist\_for\_each(iter->node, &head[iter->idx]) \                         return iter; \         } \         kfree(iter); \  \         return NULL; \ } \ \ static void \_##head##\_seq\_stop(struct seq\_file \*m, void \*p) \ { \         kfree(p); \         unlock; \ } \  \ static int \_##head##\_seq\_show(struct seq\_file \*m, void \*p) \ { \         struct \_##head##\_seq\_iter \*iter = (struct \_##head##\_seq\_iter\*)p; \  \         return imp\_show(m, iter->node); \ } \  \ \_\_DEFINE\_FOO\_SEQ\_FOPS(prefix, head, \_##head##\_seq\_show, imp\_write)  #define DEFINE\_HLIST\_SEQ\_FOPS(head, n, lock, unlock, imp\_show, imp\_write) \         \_DEFINE\_HLIST\_SEQ\_FOPS(, head, n, lock, unlock, imp\_show, imp\_write)  #define DEFINE\_STATIC\_HLIST\_SEQ\_FOPS(head, n, lock, unlock, imp\_show, \                 imp\_write) \         \_DEFINE\_HLIST\_SEQ\_FOPS(static, head, n, lock, unlock, imp\_show, \                         imp\_write)  /\* helper for rbtree \*/ #define \_DEFINE\_RBTREE\_SEQ\_FOPS(prefix, tree, lock, unlock, imp\_show, \                 imp\_write) \ static void\* \_##tree##\_seq\_start(struct seq\_file \*m, loff\_t \*pos) \ { \         struct rb\_node \*node; \         loff\_t off = \*pos; \  \         lock; \         node = rb\_first(&tree); \         while (off-- > 0 && node != NULL) \                 node = rb\_next(node); \  \         return node; \ } \  \ static void\* \_##tree##\_seq\_next(struct seq\_file \*m, void \*p, loff\_t \*pos) \ { \         ++\*pos; \         return rb\_next((struct rb\_node\*)p); \ } \ \ static void \_##tree##\_seq\_stop(struct seq\_file \*m, void \*p) \ { \         unlock; \ } \  \ \_\_DEFINE\_FOO\_SEQ\_FOPS(prefix, tree, imp\_show, imp\_write)  #define DEFINE\_RBTREE\_SEQ\_FOPS(tree, lock, unlock, imp\_show, imp\_write) \         \_DEFINE\_RBTREE\_SEQ\_FOPS(, tree, lock, unlock, imp\_show, imp\_write)  #define DEFINE\_STATIC\_RBTREE\_SEQ\_FOPS(tree, lock, unlock, imp\_show, \                 imp\_write) \         \_DEFINE\_RBTREE\_SEQ\_FOPS(static, tree, lock, unlock, imp\_show, \                         imp\_write)  #endif /\* \_SEQ\_FILE\_HELPER\_H\_ \*/ |

应用起来比较简单，下面是用其重写过的上面的双向连接表导出代码：

|  |
| --- |
| #include <linux/kernel.h> #include <linux/module.h> #include <linux/mutex.h> #include <linux/proc\_fs.h>  #include "seq\_file\_helper.h"  static struct mutex lock; static struct list\_head head; struct my\_data {         struct list\_head list;         int value; };  static void add\_one(void) {         struct my\_data \*data;          mutex\_lock(&lock);         data = kmalloc(sizeof(\*data), GFP\_KERNEL);         if (data != NULL)                 list\_add(&data->list, &head);         mutex\_unlock(&lock); }  static ssize\_t seq\_write(struct file \*file, const char \_\_user \* buffer,                        size\_t count, loff\_t \*ppos) {         add\_one();          return count; }  static int seq\_show(struct seq\_file \*m, void \*p) {         struct my\_data \*data = list\_entry(p, struct my\_data, list);         seq\_printf(m, "value: %d\n", data->value);          return 0; }  DEFINE\_STATIC\_LIST\_SEQ\_FOPS(head, mutex\_lock(&lock), mutex\_unlock(&lock),                 seq\_show, seq\_write);  static void clean\_all(struct list\_head \*head) {         struct my\_data \*data;          while (!list\_empty(head)) {                 data = list\_entry(head->next, struct my\_data, list);                 list\_del(&data->list);                 kfree(data);         } }  static int \_\_init init(void) {         struct proc\_dir\_entry \*entry;          mutex\_init(&lock);         INIT\_LIST\_HEAD(&head);          add\_one();         add\_one();         add\_one();          entry = create\_proc\_entry("my\_data", S\_IWUSR | S\_IRUGO, NULL);         if (entry == NULL) {                 clean\_all(&head);                 return -ENOMEM;         }         entry->proc\_fops = &head\_seq\_fops;          return 0; }  static void \_\_exit fini(void) {         remove\_proc\_entry("my\_data", NULL);         clean\_all(&head); }  module\_init(init); module\_exit(fini); |