

Linux/Android开发记录

学习、记录、分享Linux/Android开发技术

目录视图

摘要视图

RSS 订阅

个人资料



liuhaoyutz

访问: 80624次

积分: 1673分

排名: 第7877名

原创: 83篇

转载: 0篇

译文: 0篇

评论: 59条

博客声明

本博客文章均为原创，欢迎转载交流。转载请注明出处，禁止用于商业目的。

博客专栏



Android应用开发学习笔记

文章: 30篇

阅读: 17067



LDD3源码分析

文章: 17篇

阅读: 29970

文章分类

LDD3源码分析 (18)

ADC驱动 (1)

触摸屏驱动 (1)

LCD驱动 (1)

Linux设备模型 (8)

USB驱动 (0)

Android架构分析 (12)

Cocos2d-x (1)

C陷阱与缺陷 (3)

Android应用开发 (30)

Linux设备驱动程序架构分析 (8)

有奖征资源，博文分享有内涵

5月推荐博文汇总

大数据读书汇--获奖名单公布

2014 CSDN博文大赛

## LDD3源码分析之时间与延迟操作

分类: LDD3源码分析

2012-03-30 21:12

1988人阅读

评论(3)

收藏

举报

struct

timer

delay

工作

function

null

作者: 刘昊昱

博客: <http://blog.csdn.net/liuhaoyutz>

编译环境: Ubuntu 10.10

内核版本: 2.6.32-38-generic-pae

LDD3源码路径: examples/misc-modules/jit.c examples/misc-modules/jiq.c

本文分析LDD3第7章的示例程序jit.c和jiq.c，并给出了解决编译jiq.c文件时出现的错误的方法。

### 一、jit.c文件分析

jit.c程序是一个综合性的演示程序，涉及操作时间和延迟的各种技术。为了使程序代码最少，jit使用动态的/proc文件方式。

按照惯例，我们从模块初始化函数jit\_init开始看：

```
[cpp]
01. 263int __init jit_init(void)
02. 264{
03. 265     create_proc_read_entry("currenttime", 0, NULL, jit_currenttime, NULL);
04. 266     create_proc_read_entry("jitbusy", 0, NULL, jit_fn, (void *)JIT_BUSY);
05. 267     create_proc_read_entry("jitsched",0, NULL, jit_fn, (void *)JIT_SCHED);
06. 268     create_proc_read_entry("jitqueue",0, NULL, jit_fn, (void *)JIT_QUEUE);
07. 269     create_proc_read_entry("jitschedto", 0, NULL, jit_fn, (void *)JIT_SCHEDTO);
08. 270
09. 271     create_proc_read_entry("jitimer", 0, NULL, jit_timer, NULL);
10. 272     create_proc_read_entry("jitasklet", 0, NULL, jit_tasklet, NULL);
11. 273     create_proc_read_entry("jitasklethi", 0, NULL, jit_tasklet, (void *)1);
12. 274
13. 275     return 0; /* success */
14. 276}
```

jit\_init共调用了8次create\_proc\_read\_entry函数，我们在《LDD3源码分析之调试技术》一文中分析过这个函数，该函数用来创建一个只读的/proc入口项，其函数原型如下：

```
[cpp]
01. struct proc_dir_entry *create_proc_read_entry(const char *name,
02. mode_t mode,
03. struct proc_dir_entry *base,
04. read_proc_t *read_proc,
05. void *data)
```

<http://blog.csdn.net/liuhaoyutz/article/details/7412931>

1/20

最新评论

- LDD3源码分析之内存映射  
wzw88486969:  
@fjlhlong:unsigned long offset  
= vma->vm\_pgoff <v...
- Linux设备驱动程序架构分析之I2  
teamos: 看了你的i2c的几篇文  
章，真是受益匪浅，虽然让自己  
写还是ie不出来。非常感谢
- LDD3源码分析之块设备驱动程序  
elecfan2011: 感谢楼主的精彩讲  
解，受益匪浅啊！
- LDD3源码分析之slab高速缓存  
donghuwuwei: 省去了不少修改  
的时间，真是太好了
- LDD3源码分析之时间与延迟操作  
donghuwuwei: jit.c代码需要加上  
一个头文件。
- LDD3源码分析之slab高速缓存  
捧灰: 今天学到这里了，可是为什  
么我没有修改源码一遍就通过了  
额。。。内核版本是2.6.18-  
53.el5-x...
- LDD3源码分析之字符设备驱动程  
捧灰: 参照楼主的博客在自学~谢  
谢楼主！
- LDD3源码分析之调试技术  
fantasyhujian: 分析的很清楚，  
赞一个！
- LDD3源码分析之字符设备驱动程  
fantasyhujian: 有时间再好好读  
读，真的分析的不错！
- LDD3源码分析之hello.c与Makef  
fantasyhujian: 写的很详细，对  
初学者很有帮助！！

阅读排行

- LDD3源码分析之字符设: (3143)
- LDD3源码分析之hello.c: (2701)
- S3C2410驱动分析之LCI (2527)
- Linux设备模型分析之kse (2435)
- LDD3源码分析之内存映! (2336)
- LDD3源码分析之与硬件: (2333)
- Android架构分析之Andrc (2093)
- LDD3源码分析之时间与: (1987)
- LDD3源码分析之poll分析 (1972)
- S3C2410驱动分析之AD (1948)

评论排行

- LDD3源码分析之字符设: (12)
- S3C2410驱动分析之触指 (7)
- LDD3源码分析之内存映! (5)
- LDD3源码分析之hello.c: (4)
- Linux设备模型分析之kot (4)
- LDD3源码分析之slab高: (4)
- S3C2410驱动分析之LCI (3)
- LDD3源码分析之阻塞型I (3)
- LDD3源码分析之时间与: (3)
- LDD3源码分析之poll分析 (2)

文章存档

- 2014年06月 (1)
- 2014年05月 (4)
- 2014年04月 (1)

name是要创建的文件名称；

mode是该文件的保护掩码(传入0表示使用系统默认值)；

base指定该文件所在的目录(如果base为NULL，则该文件将创建在/proc根目录下)；

read\_proc是实现该文件的读操作的函数；

data是传递给read\_proc的参数。

这里我们重点看read\_proc函数。为了创建一个只读的/proc文件，驱动程序必须实现一个函数，用于在读取文件时生成数据，这个函数称为read\_proc。当某个进程读取这个/proc文件时(使用read系统调用)，就会调用相应驱动程序的read\_proc函数。

read\_proc函数的原型如下：

```
[cpp]
01. int (*read_proc)(char *page, char **start, off_t off, int count, int *eof, void *data)
```

这个函数的参数比较难理解，这里我偷下懒，直接把文档Linux Kernel Procfs Guide上对这个函数的解释复制过来，大家自己看。另外，LDD3中文版，翻译上有许多不对的地方，建议大家看英文版的描述。

The read function is a call back function that allows userland processes to read data from the kernel.

The read function should write its information into the *page*. For proper use, the function should start writing at an offset of *off* in *page* and write at most *count* bytes, but because most read functions are quite simple and only return a small amount of information, these two parameters are usually ignored (it breaks pagers like more and less, but cat still works).

If the *off* and *count* parameters are properly used, *eof* should be used to signal that the end of the file has been reached by writing 1 to the memory location *eof* points to.

The parameter *start* doesn't seem to be used anywhere in the kernel. The *data* parameter can be used to create a single call back function for several files.

The read\_func function must return the number of bytes written into the *page*.

理解了create\_proc\_read\_entry和read\_proc函数，我们就可以来看jit\_init函数都完成了哪些工作了。8次调用create\_proc\_read\_entry函数，也就是说，jit\_init创建了8个/proc入口项，文件名、read\_proc函数以及传递给read\_proc函数的参数的如下：

/proc/currentime	jit_currentime	NULL
/proc/jitbusy	jit_fn	JIT_BUSY
/proc/jitsched	jit_fn	JIT_SCHED
/proc/jitqueue	jit_fn	JIT_QUEUE
/proc/jitschedto	jit_fn	JIT_SCHEDTO
/proc/jitimer	jit_timer	NULL
/proc/jitasklet	jit_tasklet	NULL
/proc/jitasklethi	jit_tasklet	1

创建了这些模块入口项之后，jit模块的初始化工作就完成了。下面我们按照LDD3讲的顺序，依次看每个/proc入口项完成什么工作。

2014年01月 (1)  
2013年12月 (6)

展开

文章搜索

推荐文章

首先来看/proc/currenttime，它对应的read\_proc函数是jit\_currenttime，其代码如下：

```
[cpp]
01.  91/*
02.  92 * This file, on the other hand, returns the current time forever
03.  93 */
04.  94int jit_currenttime(char *buf, char **start, off_t offset,
05.  95                    int len, int *eof, void *data)
06.  96{
07.  97     struct timeval tv1;
08.  98     struct timespec tv2;
09.  99     unsigned long j1;
10. 100     u64 j2;
11. 101
12. 102     /* get them four */
13. 103     j1 = jiffies;
14. 104     j2 = get_jiffies_64();
15. 105     do_gettimeofday(&tv1);
16. 106     tv2 = current_kernel_time();
17. 107
18. 108     /* print */
19. 109     len=0;
20. 110     len += sprintf(buf,"0x%08lx 0x%016Lx %10i.%06i\n"
21. 111                  "%40i.%09i\n",
22. 112                  j1, j2,
23. 113                  (int) tv1.tv_sec, (int) tv1.tv_usec,
24. 114                  (int) tv2.tv_sec, (int) tv2.tv_nsec);
25. 115     *start = buf;
26. 116     return len;
27. 117}
```

jit\_currenttime函数以多种形式返回当前时间。

97 - 98行，定义了一个timeval类型的变量tv1和一个timespec类型的变量tv2，这两个结构体都定义在linux/time.h中：

```
[cpp]
01.  struct timeval {
02.      time_t      tv_sec;      /* seconds */
03.      suseconds_t tv_usec;     /* microseconds */
04.  };
05.
06.  struct timespec {
07.      time_t      tv_sec;      /* seconds */
08.      long        tv_nsec;     /* nanoseconds */
09.  };
```

用户空间表述时间就是使用这两个结构体，timeval使用秒和毫秒，timespec使用秒和纳秒，timeval出现的更早，也更常用。

103行，将当前jiffies值保存在j1中，注意，j1是unsigned long类型变量。

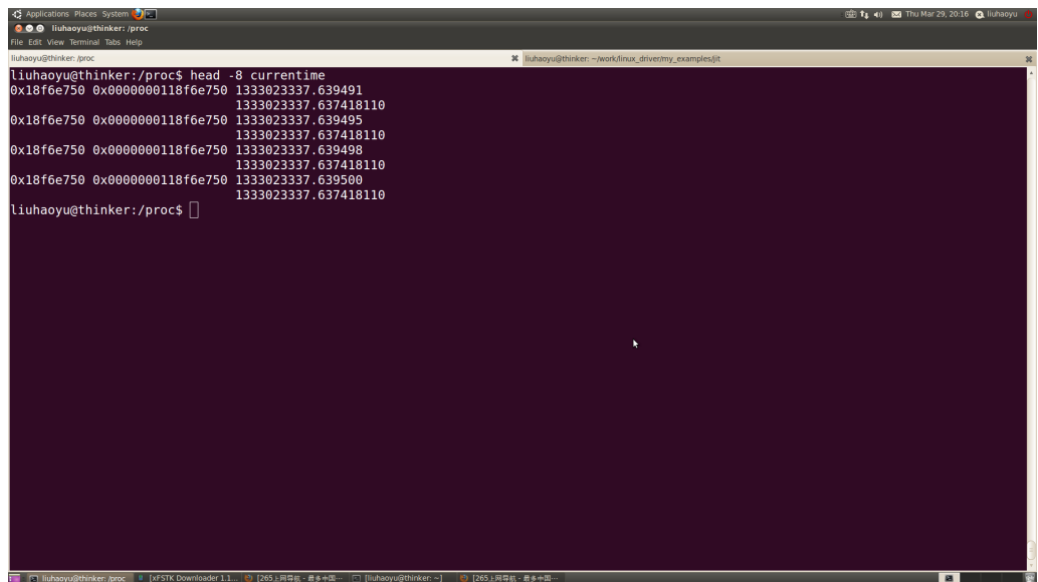
104行，调用get\_jiffies\_64函数，取得64位的jiffies值。

105行，调用do\_gettimeofday函数，把当前jiffies值转换为timeval类型保存在tv1中。

106行，调用current\_kernel\_time函数取得当前jiffies值对应的timespec类型，保存在tv2中。

110 - 114行，将4种形式的时间值保存在buf指向的内存页中。

由上面的分析可见，当读/proc/currenttime文件时，会打印4种形式的时间值。在我的机器上测试现象如下图所示：



```
liuhaoyu@thinker:~$ head -8 currentime
0x18f6e750 0x0000000118f6e750 1333023337.639491
0x18f6e750 0x0000000118f6e750 1333023337.637418110
0x18f6e750 0x0000000118f6e750 1333023337.639495
0x18f6e750 0x0000000118f6e750 1333023337.637418110
0x18f6e750 0x0000000118f6e750 1333023337.639498
0x18f6e750 0x0000000118f6e750 1333023337.637418110
0x18f6e750 0x0000000118f6e750 1333023337.639500
0x18f6e750 0x0000000118f6e750 1333023337.637418110
liuhaoyu@thinker:~$
```

接下来我们看jit\_fn函数，/proc/jitbusy、/proc/jitsched、/proc/jitqueue、/proc/jitschedto四个文件的read\_proc函数都是jit\_fn，所不同的是，读这四个文件时，会分别传递给jit\_fn四个不同的参数JIT\_BUSY、JIT\_SCHED、JIT\_QUEUE、JIT\_SCHEDTO。函数代码如下：

```
[cpp]
01. 44/* use these as data pointers, to implement four files in one function */
02. 45enum jit_files {
03. 46     JIT_BUSY,
04. 47     JIT_SCHED,
05. 48     JIT_QUEUE,
06. 49     JIT_SCHEDTO
07. 50};
08. 51
09. 52/*
10. 53 * This function prints one line of data, after sleeping one second.
11. 54 * It can sleep in different ways, according to the data pointer
12. 55 */
13. 56int jit_fn(char *buf, char **start, off_t offset,
14. 57           int len, int *eof, void *data)
15. 58{
16. 59     unsigned long j0, j1; /* jiffies */
17. 60     wait_queue_head_t wait;
18. 61
19. 62     init_waitqueue_head (&wait);
20. 63     j0 = jiffies;
21. 64     j1 = j0 + delay;
22. 65
23. 66     switch((long)data) {
24. 67         case JIT_BUSY:
25. 68             while (time_before(jiffies, j1))
26. 69                 cpu_relax();
27. 70             break;
28. 71         case JIT_SCHED:
29. 72             while (time_before(jiffies, j1)) {
30. 73                 schedule();
31. 74             }
32. 75             break;
33. 76         case JIT_QUEUE:
34. 77             wait_event_interruptible_timeout(wait, 0, delay);
35. 78             break;
36. 79         case JIT_SCHEDTO:
37. 80             set_current_state(TASK_INTERRUPTIBLE);
38. 81             schedule_timeout (delay);
39. 82             break;
40. 83     }
41. 84     j1 = jiffies; /* actual value after we delayed */
42. 85
43. 86     len = sprintf(buf, "%9li %9li\n", j0, j1);
44. 87     *start = buf;
45. 88     return len;
46. 89}
```

44 - 50行，定义了JIT\_BUSY、JIT\_SCHED、JIT\_QUEUE、JIT\_SCHEDTO四个枚举值，分别对应0，1，2，3。

59行，定义了两个unsigned long类型变量j0, j1，用来保存jiffies值。

60 - 62行，定义并初始化了等待队列头wait。

63行，将j0设置为当前jiffies值。

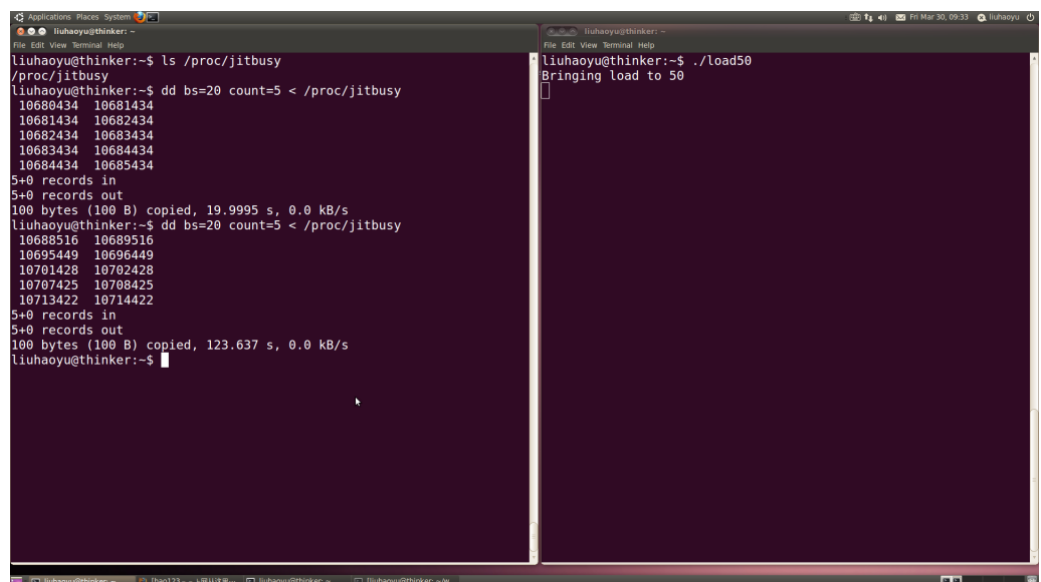
64行，将j1设置为j0 + delay。delay默认值在37行定义为HZ，也可以在命令行指定。

67 - 70行，如果传递给jit\_fn的参数是JIT\_BUSY，也就是说用户在读/proc/jitbusy文件，此时采用忙等待的方式延迟delay长度的时钟滴答。time\_before(jiffies, j1)宏在jiffies小于j1的情况下为真。cpu\_relax函数不做任何事情。

因为内核头文件中将jiffies声明为volatile型变量，所以每次C代码访问它时都会重新读取它，因此该循环可以直到延迟的作用。尽管也是正确的实现，但这个忙等待会严重降低系统性能。如果我们没有将内核配置为抢占型的，那么这个循环将在延迟期间独占处理器，在j1所代表的时间到达之前，系统看起来就像死掉了一样。而如果我们将内核配置为可抢占型的，则问题不会那么严重，这是因为除非代码拥有一个锁，否则处理器的时间还可以用作他用。但是在抢占式系统中，忙等待仍然有些浪费。

更糟糕的是，如果在进入while循环之前，禁止了中断，jiffies值就得不到更新，那么while循环的判定条件就永远为真了。

下图是测试/proc/jitbusy的过程：



```
liuhaoyu@thinker:~$ ls /proc/jitbusy
/proc/jitbusy
liuhaoyu@thinker:~$ dd bs=20 count=5 < /proc/jitbusy
10680434 10681434
10681434 10682434
10682434 10683434
10683434 10684434
10684434 10685434
5+0 records in
5+0 records out
100 bytes (100 B) copied, 19.9995 s, 0.0 kB/s
liuhaoyu@thinker:~$ dd bs=20 count=5 < /proc/jitbusy
10688516 10689516
10695449 10696449
10701428 10702428
10707425 10708425
10713422 10714422
5+0 records in
5+0 records out
100 bytes (100 B) copied, 123.637 s, 0.0 kB/s
liuhaoyu@thinker:~$

liuhaoyu@thinker:~$ ./load50
Bringing load to 50
```

上图中,第一次执行“dd bs=20 count=5 < /proc/jitbusy”时,是在系统负载比较小的情况下。第二次执行“dd bs=20 count=5 < /proc/jitbusy”时, 在另外一个终端下执行了LDD3提供的load50程序, 该程序用来增加系统负载。

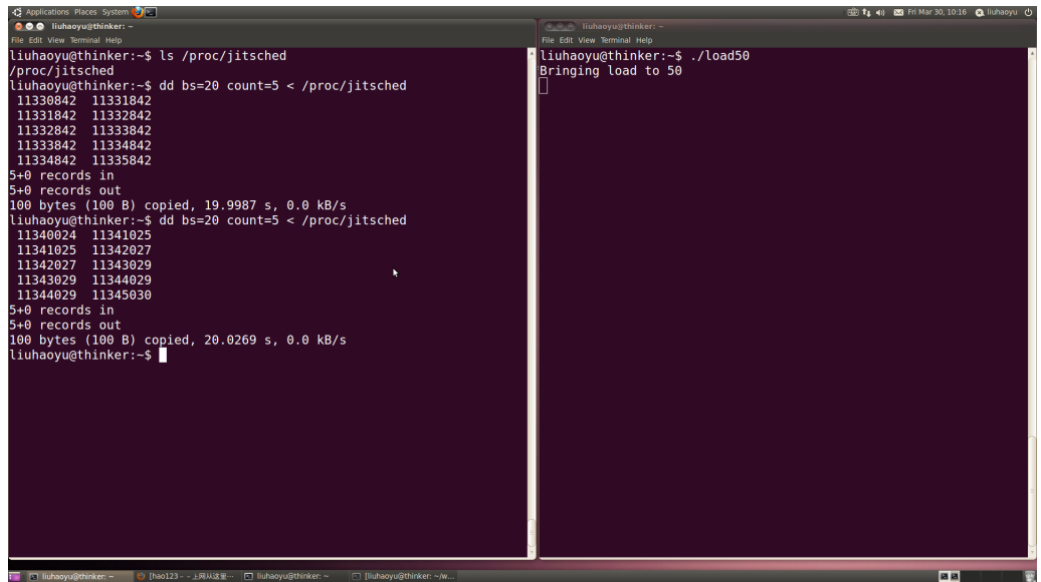
从上图中的输出内容可以看出，当系统负载较小时，每次读操作延迟恰好是1秒(1000个jiffies)，而下一次读操作会在前一个结束后立即执行。但是，当系统负载较大时，每个读操作的延迟恰好是1秒，但是下一次读操作可能会5秒后才会执行。

我们回到jit\_fn函数继续往下看。

71 - 75行，如果传递给jit\_fn的参数是JIT\_SCHED，也就是说用户在读/proc/jitsched文件，此时如果time\_before(jiffies, j1)宏为真，则调用schedule函数让出处理器。

虽然这种延迟技术在等待时放弃了CPU，但是性能仍然不理想。当前进程虽然放弃了CPU，但是它仍然在运行队列中，如果系统中只有一个可运行进程，则该进程又会立即运行，而空闲任务(进程号为0，称为swapper)从来不会运行。在系统空闲时运行swapper可以减轻处理器负载，延长处理器寿命，降低电量消耗。

下图是测试/proc/jitsched的过程：



The screenshot shows two terminal windows. The left window is titled 'liuhaoyu@thinker: ~' and shows the execution of 'dd' commands to read from '/proc/jitsched'. The first 'dd' command shows a copy time of 19.9987 s. The second 'dd' command shows a copy time of 20.0269 s. The right window is titled 'liuhaoyu@thinker: ~' and shows the execution of './load50', which brings the load to 50.

```
liuhaoyu@thinker:~$ ls /proc/jitsched
/proc/jitsched
liuhaoyu@thinker:~$ dd bs=20 count=5 < /proc/jitsched
11330842 11331842
11331842 11332842
11332842 11333842
11333842 11334842
11334842 11335842
5+0 records in
5+0 records out
100 bytes (100 B) copied, 19.9987 s, 0.0 kB/s
liuhaoyu@thinker:~$ dd bs=20 count=5 < /proc/jitsched
11340024 11341025
11341025 11342027
11342027 11343029
11343029 11344029
11344029 11345030
5+0 records in
5+0 records out
100 bytes (100 B) copied, 20.0269 s, 0.0 kB/s
liuhaoyu@thinker:~$
```

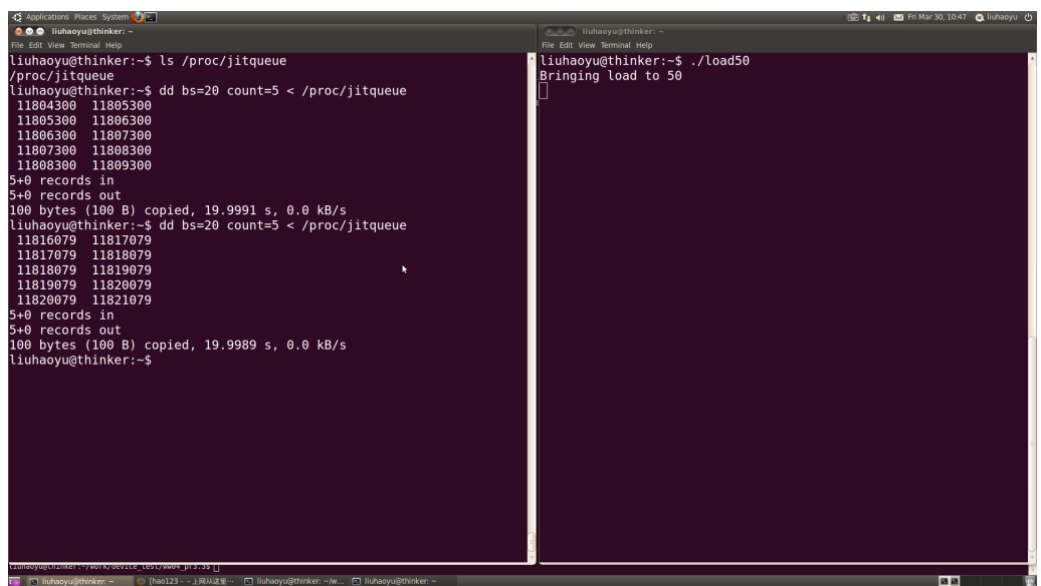
```
liuhaoyu@thinker:~$ ./load50
Bringing load to 50
```

上图中,第一次执行“dd bs=20 count=5 < /proc/jitsched”时,是在系统负载比较小的情况下。第二次执行“dd bs=20 count=5 < /proc/jitsched”时,在另外一个终端下执行了load50程序增加系统负载。可以看出,在系统负载较大时,读操作的延迟比所请求的延迟时间长几个时钟滴答,系统越忙,这个问题越突出。当一个进程使用schedule释放处理器后,不能保证在进程需要时很快就能得到处理器。

因此,除了影响系统整体性能外,使用schedule延迟的方法对驱动程序需求来讲并不安全。

我们再回到jit\_fn函数,76 - 78行,如果传递给jit\_fn的参数是JIT\_QUEUE,也就是说用户在读/proc/jitqueue文件,则调用wait\_event\_interruptible\_timeout(wait, 0, delay)函数在等待队列wait上休眠,但是会在指定的等待时间到期时返回。在到期之前,如果进程收到信号也会退出休眠返回。

下图是测试/proc/jitqueue的过程:



The screenshot shows two terminal windows. The left window is titled 'liuhaoyu@thinker: ~' and shows the execution of 'dd' commands to read from '/proc/jitqueue'. The first 'dd' command shows a copy time of 19.9991 s. The second 'dd' command shows a copy time of 19.9989 s. The right window is titled 'liuhaoyu@thinker: ~' and shows the execution of './load50', which brings the load to 50.

```
liuhaoyu@thinker:~$ ls /proc/jitqueue
/proc/jitqueue
liuhaoyu@thinker:~$ dd bs=20 count=5 < /proc/jitqueue
11804300 11805300
11805300 11806300
11806300 11807300
11807300 11808300
11808300 11809300
5+0 records in
5+0 records out
100 bytes (100 B) copied, 19.9991 s, 0.0 kB/s
liuhaoyu@thinker:~$ dd bs=20 count=5 < /proc/jitqueue
11816079 11817079
11817079 11818079
11818079 11819079
11819079 11820079
11820079 11821079
5+0 records in
5+0 records out
100 bytes (100 B) copied, 19.9989 s, 0.0 kB/s
liuhaoyu@thinker:~$
```

```
liuhaoyu@thinker:~$ ./load50
Bringing load to 50
```

从上图可以看出,即使是在高负载的系统上,对/proc/jitqueue的读取也将得到接近优化的结果。

回到jit\_fn函数,现在看79 - 82行,如果传递给jit\_fn的参数是JIT\_SCHEDTO,也就是说用户在读/proc/schedto文件,则在80行调用set\_current\_state将进程状态设置为TASK\_INTERRUPTIBLE,然后在81行调用schedule\_timeout(delay),schedule\_timeout函数让出处理器,并等待delay指定的时间。实际上wait\_event\_interruptible\_timeout函数在内部就是利用了schedule\_timeout函数。

使用schedule\_timeout方式延迟,性能与使用wait\_event\_interruptible\_timeout相似,但是可以避免声明和使用多余的等待队列。

下图是测试/proc/schedto文件的过程:

```

Applications Places System
liuhaoyu@thinker:~$ ls /proc/jitschedto
/proc/jitschedto
liuhaoyu@thinker:~$ dd bs=20 count=5 < /proc/jitschedto
11928140 11929140
11929140 11930140
11930140 11931140
11931140 11932140
11932140 11933140
5+0 records in
5+0 records out
100 bytes (100 B) copied, 19.9991 s, 0.0 kB/s
liuhaoyu@thinker:~$ dd bs=20 count=5 < /proc/jitschedto
11936340 11937340
11937340 11938340
11938340 11939340
11939340 11940340
11940340 11941340
5+0 records in
5+0 records out
100 bytes (100 B) copied, 19.9989 s, 0.0 kB/s
liuhaoyu@thinker:~$

```

从上图打印的信息可以看出，即使是在高负载的系统上，对/proc/schedto的读取也将得到接近优化的结果。

到此，jit\_fn函数我们就分析完了。下面我们来看jit\_timer函数，它对应/proc/jitimer其代码如下：

```

[cpp]
01. 155/* the /proc function: allocate everything to allow concurrency */
02. 156int jit_timer(char *buf, char **start, off_t offset,
03. 157             int len, int *eof, void *unused_data)
04. 158{
05. 159     struct jit_data *data;
06. 160     char *buf2 = buf;
07. 161     unsigned long j = jiffies;
08. 162
09. 163     data = kmalloc(sizeof(*data), GFP_KERNEL);
10. 164     if (!data)
11. 165         return -ENOMEM;
12. 166
13. 167     init_timer(&data->timer);
14. 168     init_waitqueue_head (&data->wait);
15. 169
16. 170     /* write the first lines in the buffer */
17. 171     buf2 += sprintf(buf2, "  time  delta inirq  pid  cpu command\n");
18. 172     buf2 += sprintf(buf2, "%9li %3li  %i  %6i  %i  %s\n",
19. 173                   j, 0L, in_interrupt() ? 1 : 0,
20. 174                   current->pid, smp_processor_id(), current->comm);
21. 175
22. 176     /* fill the data for our timer function */
23. 177     data->prevjiffies = j;
24. 178     data->buf = buf2;
25. 179     data->loops = JIT_ASYNC_LOOPS;
26. 180
27. 181     /* register the timer */
28. 182     data->timer.data = (unsigned long)data;
29. 183     data->timer.function = jit_timer_fn;
30. 184     data->timer.expires = j + tdelay; /* parameter */
31. 185     add_timer(&data->timer);
32. 186
33. 187     /* wait for the buffer to fill */
34. 188     wait_event_interruptible(data->wait, !data->loops);
35. 189     if (signal_pending(current))
36. 190         return -ERESTARTSYS;
37. 191     buf2 = data->buf;
38. 192     kfree(data);
39. 193     *eof = 1;
40. 194     return buf2 - buf;
41. 195}

```

159行，定义了jit\_data结构体指针变量data，jit\_data结构体定义如下：

```

[cpp]
01. 126/* This data structure used as "data" for the timer and tasklet functions */
02. 127struct jit_data {

```



```
03. 128 struct timer_list timer;
04. 129 struct tasklet_struct tlet;
05. 130 int hi; /* tasklet or tasklet_hi */
06. 131 wait_queue_head_t wait;
07. 132 unsigned long prevjiffies;
08. 133 unsigned char *buf;
09. 134 int loops;
10. 135};
```

167行，调用init\_timer初始化定时器data->timer。

168行，调用init\_waitqueue\_head初始化等待队列头data->wait。

171行，向buf2中写入标题头，同时更新buf2的值。

172行，向buf2中写入相关数据，同时更新buf2的值。

177 - 179行，初始化data->prevjiffies为当前jiffies值，data->buf指向buf2，data->loops设置为JIT\_ASYNC\_LOOPS，JIT\_ASYNC\_LOOPS在136行定义为5。

181 - 185行，设置和注册定时器dev->timer。定时器函数为jit\_timer\_fn，传递给该函数的参数为data变量，定时器等待时间为j+tdelay，最后调用add\_timer将定时器注册到系统中，此时，即启动了定时器。

188行，在data->wait上休眠等待，注意唤醒条件是data->loops变为0。

193行，将\*eof设置为1，表示读/proc/jitimer文件结束。

下面该分析定时器函数jit\_timer\_fn了，其代码如下：

```
[cpp]
01. 138void jit_timer_fn(unsigned long arg)
02. 139{
03. 140 struct jit_data *data = (struct jit_data *)arg;
04. 141 unsigned long j = jiffies;
05. 142 data->buf += sprintf(data->buf, "%9li %3li %i %6i %i %s\n",
06. 143 j, j - data->prevjiffies, in_interrupt() ? 1 : 0,
07. 144 current->pid, smp_processor_id(), current->comm);
08. 145
09. 146 if (--data->loops) {
10. 147 data->timer.expires += tdelay;
11. 148 data->prevjiffies = j;
12. 149 add_timer(&data->timer);
13. 150 } else {
14. 151 wake_up_interruptible(&data->wait);
15. 152 }
16. 153}
```

在定时器函数jit\_timer\_fn中，首先向data->buf中写入相关数据，然后判断data->loops的值是否为0，如果不为0，重新设置相关参数并再次启动定时器；如果data->loops的值变为0，则唤醒等待队列data->wait上的进程。

测试/proc/jitimer的过程如下图所示：



```

liuhaoyu@thinker:~/work/linux_driver/my_examples/jit$ cat /proc/jitimer
time delta inirq pid cpu command
3473394 0 0 6482 5 cat
3473404 10 1 0 5 swapper
3473414 10 1 0 5 swapper
3473424 10 1 0 5 swapper
3473434 10 1 0 5 swapper
3473444 10 1 0 5 swapper
liuhaoyu@thinker:~/work/linux_driver/my_examples/jit$

```

下面我们来看jit\_tasklet函数，该函数对应的文件是/proc/jitasklet和/proc/jitasklethi，代码如下：

```

[cpp]

01. 216/* the /proc function: allocate everything to allow concurrency */
02. 217int jit_tasklet(char *buf, char **start, off_t offset,
03. 218               int len, int *eof, void *arg)
04. 219{
05. 220     struct jit_data *data;
06. 221     char *buf2 = buf;
07. 222     unsigned long j = jiffies;
08. 223     long hi = (long)arg;
09. 224
10. 225     data = kmalloc(sizeof(*data), GFP_KERNEL);
11. 226     if (!data)
12. 227         return -ENOMEM;
13. 228
14. 229     init_waitqueue_head (&data->wait);
15. 230
16. 231     /* write the first lines in the buffer */
17. 232     buf2 += sprintf(buf2, "  time  delta inirq  pid  cpu command\n");
18. 233     buf2 += sprintf(buf2, "%9li %3li  %i  %6i  %i  %s\n",
19. 234                   j, 0L, in_interrupt() ? 1 : 0,
20. 235                   current->pid, smp_processor_id(), current->comm);
21. 236
22. 237     /* fill the data for our tasklet function */
23. 238     data->prevjiffies = j;
24. 239     data->buf = buf2;
25. 240     data->loops = JIT_ASYNC_LOOPS;
26. 241
27. 242     /* register the tasklet */
28. 243     tasklet_init(&data->tlet, jit_tasklet_fn, (unsigned long)data);
29. 244     data->hi = hi;
30. 245     if (hi)
31. 246         tasklet_hi_schedule(&data->tlet);
32. 247     else
33. 248         tasklet_schedule(&data->tlet);
34. 249
35. 250     /* wait for the buffer to fill */
36. 251     wait_event_interruptible(data->wait, !data->loops);
37. 252
38. 253     if (signal_pending(current))
39. 254         return -ERESTARTSYS;
40. 255     buf2 = data->buf;
41. 256     kfree(data);
42. 257     *eof = 1;
43. 258     return buf2 - buf;
44. 259}

```

jit\_tasklet的实现与前面分析的jit\_timer很类似，相同的地方这里我们不仔细分析了，这里重点看一下两个函数不同的地方。

243行，调用tasklet\_init初始化data->tlet，指定tasklet函数为jit\_tasklet\_fn，交将data作为参数传递给jit\_tasklet\_fn。

245 - 246行, 如果hi为1, 则使用tasklet\_hi\_schedule函数以高优先级调度data->tlet执行。当软件中断处理程序运行时, 它会在处理其他软件中断(包括普通的tasklet)之前, 处理高优先级的tasklet。

247 - 248行, 如果hi为0, 则使用tasklet\_schedule调度data->tlet执行。

251行, 在data->wait上休眠等待, 注意唤醒条件是data->loops变为0。

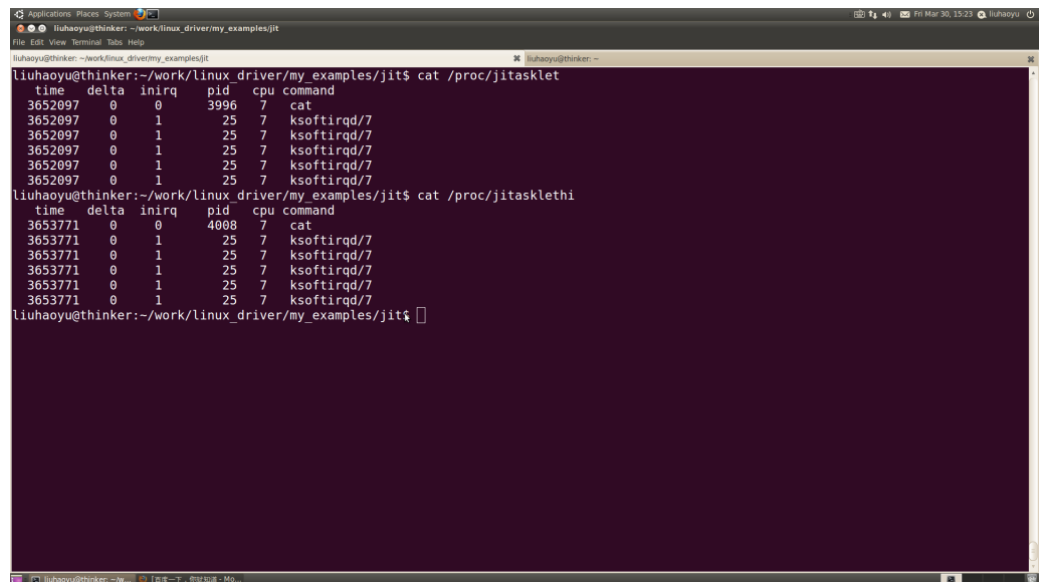
257行, 将\*eof设置为1, 表示读/proc/ jitasklet或/proc/ jitasklethi文件结束。

下面看tasklet函数jit\_tasklet\_fn:

```
[cpp]
01. 197 void jit_tasklet_fn(unsigned long arg)
02. 198 {
03. 199     struct jit_data *data = (struct jit_data *)arg;
04. 200     unsigned long j = jiffies;
05. 201     data->buf += sprintf(data->buf, "%9li %3li    %i    %6i    %i    %s\n",
06. 202                         j, j - data->prevjiffies, in_interrupt() ? 1 : 0,
07. 203                         current->pid, smp_processor_id(), current->comm);
08. 204
09. 205     if (--data->loops) {
10. 206         data->prevjiffies = j;
11. 207         if (data->hi)
12. 208             tasklet_hi_schedule(&data->tlet);
13. 209         else
14. 210             tasklet_schedule(&data->tlet);
15. 211     } else {
16. 212         wake_up_interruptible(&data->wait);
17. 213     }
18. 214 }
```

该函数的功能与前面介绍的定时器函数jit\_timer\_fn非常像, 只不过把调度定时器的操作换成了调度tasklet的函数。

下图是测试/proc/ jitasklet或/proc/ jitasklethi的过程:



```
liuhaoyu@thinker:~/work/linux_driver/my_examples/jit$ cat /proc/jitasklet
time delta inirq pid cpu command
3652097 0 0 3996 7 cat
3652097 0 1 25 7 ksoftirqd/7
3652097 0 1 25 7 ksoftirqd/7
3652097 0 1 25 7 ksoftirqd/7
3652097 0 1 25 7 ksoftirqd/7
3652097 0 1 25 7 ksoftirqd/7
3652097 0 1 25 7 ksoftirqd/7
liuhaoyu@thinker:~/work/linux_driver/my_examples/jit$ cat /proc/jitasklethi
time delta inirq pid cpu command
3653771 0 0 4008 7 cat
3653771 0 1 25 7 ksoftirqd/7
3653771 0 1 25 7 ksoftirqd/7
3653771 0 1 25 7 ksoftirqd/7
3653771 0 1 25 7 ksoftirqd/7
3653771 0 1 25 7 ksoftirqd/7
liuhaoyu@thinker:~/work/linux_driver/my_examples/jit$
```

至此, jit.c文件我们就全部分析完了。

## 二、jiq.c文件分析

LDD3第7章涉及的代码还有一个jiq.c文件, 这个文件主要演示了等待队列的用法。下面是jiq模块初始化函数jiq\_init的代码:

```
[cpp]
01. 240 static int jiq_init(void)
02. 241 {
03. 242
04. 243     /* this line is in jiq_init() */
```

```

05. 244 INIT_WORK(&jiq_work, jiq_print_wq, &jiq_data);
06. 245
07. 246 create_proc_read_entry("jiqwq", 0, NULL, jiq_read_wq, NULL);
08. 247 create_proc_read_entry("jiqwqdelay", 0, NULL, jiq_read_wq_delayed, NULL);
09. 248 create_proc_read_entry("jitimer", 0, NULL, jiq_read_run_timer, NULL);
10. 249 create_proc_read_entry("jiqtasklet", 0, NULL, jiq_read_tasklet, NULL);
11. 250
12. 251 return 0; /* succeed */
13. 252}

```

244行，初始化一个工作jiq\_work，其工作函数是jiq\_print\_wq，jiq\_data是传递给该函数的参数。jiq\_work是work\_struct类型的变量，struct work\_struct在linux/workqueue.h文件中定义如下：

```

[cpp]
01. struct work_struct {
02.     atomic_long_t data;
03.     #define WORK_STRUCT_PENDING 0 /* T if work item pending execution */
04.     #define WORK_STRUCT_FLAG_MASK (3UL)
05.     #define WORK_STRUCT_WQ_DATA_MASK (~WORK_STRUCT_FLAG_MASK)
06.     struct list_head entry;
07.     work_func_t func;
08.     #ifdef CONFIG_LOCKDEP
09.     struct lockdep_map lockdep_map;
10.     #endif
11. };

```

jiq\_data是一个全局变量，其定义如下：

```

[cpp]
01. 63static struct clientdata {
02.     64 int len;
03.     65 char *buf;
04.     66 unsigned long jiffies;
05.     67 long delay;
06.     68} jiq_data;

```

246 - 249行，创建4个/proc只读接口，它们分别是/proc/jiqwq、/proc/jiqwqdelay、/proc/jitimer、/proc/jiqtasklet，这4个文件对应的read\_proc函数分别是jiq\_read\_wq、jiq\_read\_wq\_delayed、jiq\_read\_run\_timer、jiq\_read\_tasklet。

首先来看jiq\_read\_wq函数的实现：

```

[cpp]
01. 129static int jiq_read_wq(char *buf, char **start, off_t offset,
02. 130 int len, int *eof, void *data)
03. 131{
04. 132     DEFINE_WAIT(wait);
05. 133
06. 134     jiq_data.len = 0; /* nothing printed, yet */
07. 135     jiq_data.buf = buf; /* print in this place */
08. 136     jiq_data.jiffies = jiffies; /* initial time */
09. 137     jiq_data.delay = 0;
10. 138
11. 139     prepare_to_wait(&jiq_wait, &wait, TASK_INTERRUPTIBLE);
12. 140     schedule_work(&jiq_work);
13. 141     schedule();
14. 142     finish_wait(&jiq_wait, &wait);
15. 143
16. 144     *eof = 1;
17. 145     return jiq_data.len;
18. 146}

```

jiq\_read\_wq函数中使用的是共享工作队列，而不是进程专用的工作队列，共享工作队列更节约系统资源。该函数的休眠采用的是手工休眠。

132行，使用DEFINE\_WAIT宏定义一个等待队列入口wait。

134 - 137行，对jiq\_data进行初始化。

139行，调用prepare\_to\_wait函数将wait加入到等待队列jiq\_wait中，并把进程状态设置为TASK\_INTERRUPTIBLE。

140行，调用schedule\_work将jiq\_work提交到系统提供的共享工作队列中。

141行, 调用schedule函数, 让出处理器, 此后, 进程就在等待队列jiq\_wait中休眠。

142行, 休眠被唤醒后, 调用finish\_wait做一些清理工作。

144行, 将\*eof设置为1, 表明已经读到/proc/jiqwq的结束处。

把工作jiq\_work提交给共享工作队列后, 在此后的某一时间, 工作函数jiq\_print\_wq就会执行, 该函数代码如下:

```
[cpp]
01. 111/*
02. 112 * Call jiq_print from a work queue
03. 113 */
04. 114static void jiq_print_wq(void *ptr)
05. 115{
06. 116     struct clientdata *data = (struct clientdata *) ptr;
07. 117
08. 118     if (! jiq_print (ptr))
09. 119         return;
10. 120
11. 121     if (data->delay)
12. 122         schedule_delayed_work(&jiq_work, data->delay);
13. 123     else
14. 124         schedule_work(&jiq_work);
15. 125}
```

工作函数jiq\_print\_wq的核心代码在jiq\_print函数中, 如果该函数返回0, 则退出。如果返回非0值, 说明还需要再次调用该工作。如果jiq\_print函数返回非0值, 根据data->delay的值是否为0, 分别采用schedule\_delayed\_work和schedule\_work, 以延迟和非延迟的方式重新调度工作。

jiq\_print函数定义如下:

```
[cpp]
01. 78/*
02. 79 * Do the printing; return non-zero if the task should be rescheduled.
03. 80 */
04. 81static int jiq_print(void *ptr)
05. 82{
06. 83     struct clientdata *data = ptr;
07. 84     int len = data->len;
08. 85     char *buf = data->buf;
09. 86     unsigned long j = jiffies;
10. 87
11. 88     if (len > LIMIT) {
12. 89         wake_up_interruptible(&jiq_wait);
13. 90         return 0;
14. 91     }
15. 92
16. 93     if (len == 0)
17. 94         len = sprintf(buf, "    time delta preempt  pid cpu command\n");
18. 95     else
19. 96         len = 0;
20. 97
21. 98     /* intr_count is only exported since 1.3.5, but 1.99.4 is needed anyways */
22. 99     len += sprintf(buf+len, "%9li %4li    %3i %5i %3i %s\n",
23. 100         j, j - data->jiffies,
24. 101         preempt_count(), current->pid, smp_processor_id(),
25. 102         current->comm);
26. 103
27. 104     data->len += len;
28. 105     data->buf += len;
29. 106     data->jiffies = j;
30. 107     return 1;
31. 108}
```

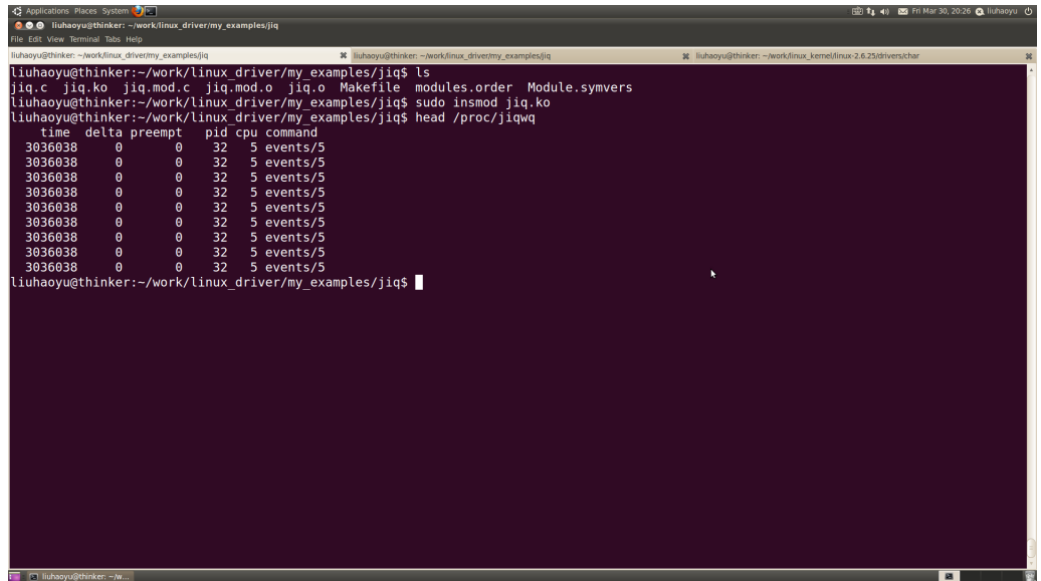
88 - 91行, 如果打印的信息已经超过了LIMIT()的限制, 则唤醒等待队列jiq\_wait上的进程, 并返回0, 表示不需要再次调度该工作。

93 - 102行, 打印相关信息。

104 - 107行, 更新jiq\_data的信息。

108行, 返回1, 表明还需要再次调度该工作。

下图是对/proc/jiqwq的测试过程:



```
liuhaoyu@thinker:~/work/linux_driver/my_examples/jiq$ ls
jiq.c  jiq.ko  jiq.mod.c  jiq.mod.o  jiq.o  Makefile  modules.order  Module.symvers
liuhaoyu@thinker:~/work/linux_driver/my_examples/jiq$ sudo insmod jiq.ko
liuhaoyu@thinker:~/work/linux_driver/my_examples/jiq$ head /proc/jiqwq
time  delta  preempt  pid  cpu  command
3036038  0  0  32  5  events/5
3036038  0  0  32  5  events/5
3036038  0  0  32  5  events/5
3036038  0  0  32  5  events/5
3036038  0  0  32  5  events/5
3036038  0  0  32  5  events/5
3036038  0  0  32  5  events/5
3036038  0  0  32  5  events/5
3036038  0  0  32  5  events/5
3036038  0  0  32  5  events/5
liuhaoyu@thinker:~/work/linux_driver/my_examples/jiq$
```

下面我们来看jiq\_read\_wq\_delayed函数, 其代码如下:

```
[cpp]
01. 149static int jiq_read_wq_delayed(char *buf, char **start, off_t offset,
02. 150                                int len, int *eof, void *data)
03. 151{
04. 152     DEFINE_WAIT(wait);
05. 153
06. 154     jiq_data.len = 0;           /* nothing printed, yet */
07. 155     jiq_data.buf = buf;        /* print in this place */
08. 156     jiq_data.jiffies = jiffies; /* initial time */
09. 157     jiq_data.delay = delay;
10. 158
11. 159     prepare_to_wait(&jiq_wait, &wait, TASK_INTERRUPTIBLE);
12. 160     schedule_delayed_work(&jiq_work, delay);
13. 161     schedule();
14. 162     finish_wait(&jiq_wait, &wait);
15. 163
16. 164     *eof = 1;
17. 165     return jiq_data.len;
18. 166}
```

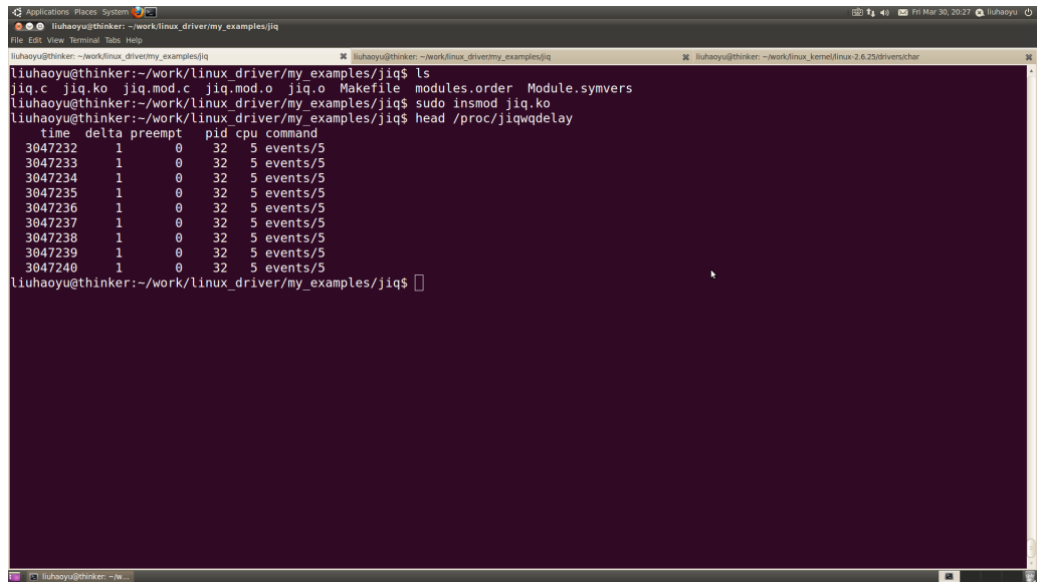
这个函数与前面分析的jiq\_read\_wq非常相似, 区别是这里采用延迟方式调度工作。这里我只看两个函数不同的地方, 如果有不明白的, 参考jiq\_read\_wq理解。

157行, 因为要采用延迟方式, 设置jiq\_data.delay的值为delay, 其默认值为1, 可以通过命令行传递参数修改。

160行, schedule\_delayed\_work以延迟模式将工作提交到工作队列。

把工作jiq\_work提交给共享工作队列后, 在此后的某一时间, 工作函数jiq\_print\_wq就会执行, 该函数我们前面已经分析过, /proc/jiqwqdelay与/proc/jiqwq的区别是因为指定了jiq\_data.delay的值为非0值, 所以121行, 如果需要再次调度工作, 还是用延迟模式调度。

下图是测试/proc/jiqwqdelay的过程:



下面我们来看jiq\_read\_run\_timer函数，这个函数使用的是定时器，其代码如下：

```
[cpp]
01. 212static int jiq_read_run_timer(char *buf, char **start, off_t offset,
02. 213                               int len, int *eof, void *data)
03. 214{
04. 215
05. 216     jiq_data.len = 0;           /* prepare the argument for jiq_print() */
06. 217     jiq_data.buf = buf;
07. 218     jiq_data.jiffies = jiffies;
08. 219
09. 220     init_timer(&jiq_timer);      /* init the timer structure */
10. 221     jiq_timer.function = jiq_timedout;
11. 222     jiq_timer.data = (unsigned long)&jiq_data;
12. 223     jiq_timer.expires = jiffies + HZ; /* one second */
13. 224
14. 225     jiq_print(&jiq_data); /* print and go to sleep */
15. 226     add_timer(&jiq_timer);
16. 227     interruptible_sleep_on(&jiq_wait); /* RACE */
17. 228     del_timer_sync(&jiq_timer); /* in case a signal woke us up */
18. 229
19. 230     *eof = 1;
20. 231     return jiq_data.len;
21. 232}
```

有了前面分析的基础，我们看这个函数应该比较轻松，因为大部分代码和前面分析过的代码是一样的。

216 - 218行，初始化jiq\_data。

220 - 223行，初始化定时器jiq\_timer，定时器函数是jiq\_timedout。

225行，调用jiq\_print打印标题栏和一行信息。

226行，启动定时器。

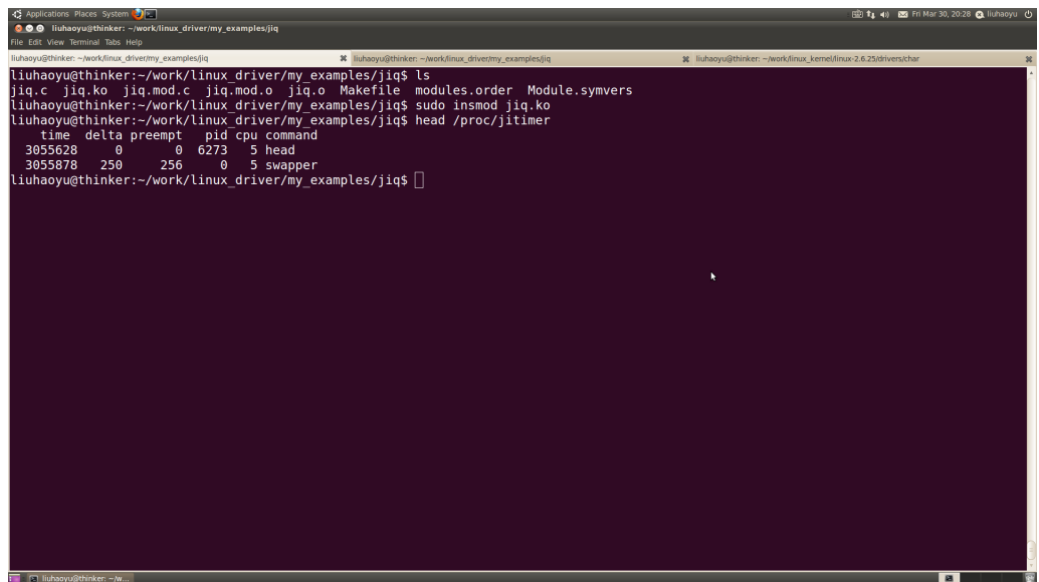
227行，进程在jiq\_wait上休眠等待。

定时器函数是jiq\_timedout的代码如下：

```
[cpp]
01. 205static void jiq_timedout(unsigned long ptr)
02. 206{
03. 207     jiq_print((void *)ptr);      /* print a line */
04. 208     wake_up_interruptible(&jiq_wait); /* awake the process */
05. 209}
```

该定时器函数只执行一次，207行打印一行信息，208行唤醒等待进程。

下图是测试/proc/jitimer的过程：



下面我们看jiq\_read\_tasklet函数，这个函数使用的是tasklet，其代码如下：

```
[cpp]
01. 182static int jiq_read_tasklet(char *buf, char **start, off_t offset, int len,
02. 183                             int *eof, void *data)
03. 184{
04. 185     jiq_data.len = 0;           /* nothing printed, yet */
05. 186     jiq_data.buf = buf;         /* print in this place */
06. 187     jiq_data.jiffies = jiffies; /* initial time */
07. 188
08. 189     tasklet_schedule(&jiq_tasklet);
09. 190     interruptible_sleep_on(&jiq_wait); /* sleep till completion */
10. 191
11. 192     *eof = 1;
12. 193     return jiq_data.len;
13. 194}
```

189行，调度jiq\_tasklet。

190行，进程在jiq\_wait上休眠。

jiq\_tasklet在第75行定义：

```
[cpp]
01. 75static DECLARE_TASKLET(jiq_tasklet, jiq_print_tasklet, (unsigned long)&jiq_data);
```

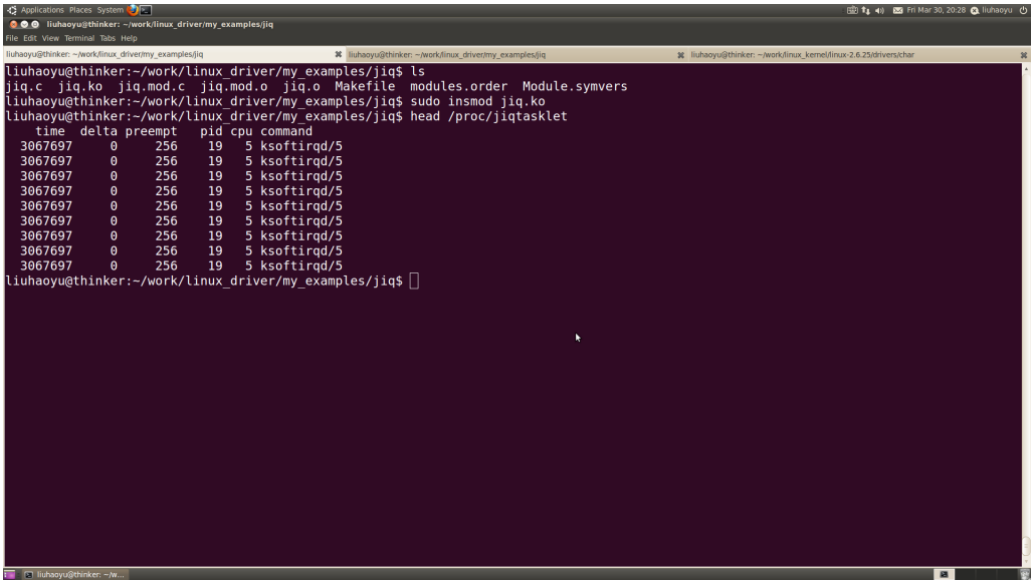
tasklet函数是jiq\_print\_tasklet，其定义如下：

```
[cpp]
01. 171/*
02. 172 * Call jiq_print from a tasklet
03. 173 */
04. 174static void jiq_print_tasklet(unsigned long ptr)
05. 175{
06. 176     if (jiq_print ((void *) ptr))
07. 177         tasklet_schedule (&jiq_tasklet);
08. 178}
```

调用jiq\_print打印信息，如果jiq\_print返回非0值，重新调度tasklet。

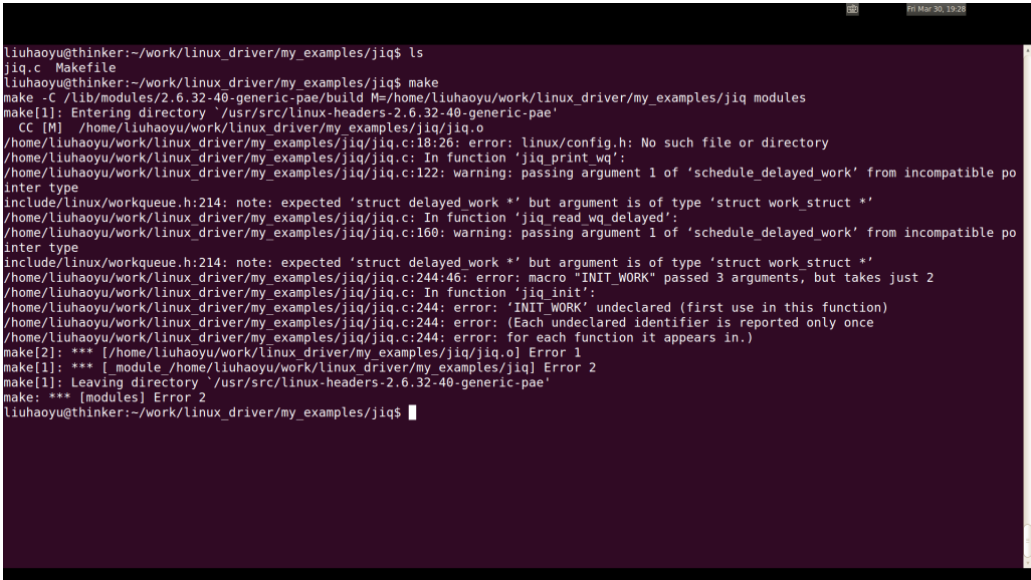
测试/proc/jiqtasklet的过程如下图所示：



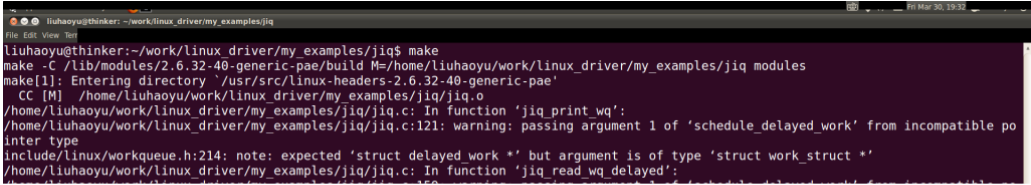


三、编译jiq模块时出现的问题

在编译jiq模块过程中，出现如下图所示错误：



这是因为linux/config.h在2.6.32-38-generic-pae内核中不存在了，解决方法是直接删掉jiq的第18行即可。再次编译，出现如下错误：



这是因为在2.6.32-38-generic-pae内核中，INIT\_WORK宏的定义发生了变化，只能接受两个参数，但是LDD3传递了3个参数。解决办法是把243行

```
[cpp]
01. 243 INIT_WORK(&jiq_work, jiq_print_wq, &jiq_data);
```

替换为

```
[cpp]
01. 243 INIT_WORK(&jiq_work, jiq_print_wq);
```

这样替换完成后，还有几个连锁反应需要处理。因为LDD3中的本意是把jiq\_data作为参数传递给jiq\_print\_wq函数。现在新的INIT\_WORK中无法传递参数了，所以我们需要修改jiq\_print\_wq函数，手动把jiq\_data传递过去。修改后的jiq\_print\_wq代码如下：

```
[cpp]
01. 113static void jiq_print_wq(struct work_struct *work)
02. 114{
03. 115     struct clientdata *data = (struct clientdata *) &jiq_data;
04. 116
05. 117     if (! jiq_print (&jiq_data))
06. 118         return;
07. 119
08. 120     if (data->delay)
09. 121         schedule_delayed_work(&jiq_work, data->delay);
10. 122     else
11. 123         schedule_work(&jiq_work);
12. 124}
```

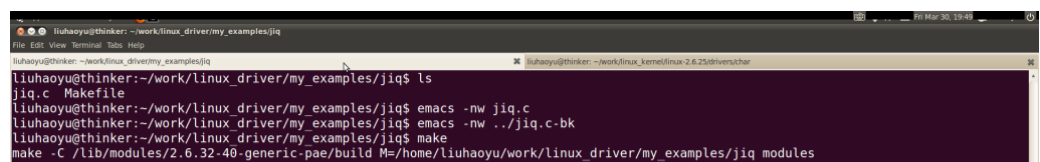
与原来相比，共修改了3个地方。

113行，参数类型改为work\_struct结构指针，这是新内核定义的参数类型。

115行，直接使用全局变量jiq\_data，原来是通过参数传递的。

117行，直接使用全局变量jiq\_data，原来是通过参数传递的。

修改后，再编译，能编译通过，但仍有问题，如下图所示：



这是因为，在2.6.32-38-generic-pae内核中，schedule\_delayed\_work函数的定义发生了变化，现在其函数原型是

```
[cpp]
01. int schedule_delayed_work(struct delayed_work *dwork, unsigned long delay)
```

而在LDD3使用的2.6.10版本的内核中，其函数原型是：

[cpp]

```
01. int fastcall schedule_delayed_work(struct work_struct *work, unsigned long delay)
```

所以要在新的内核上执行schedule\_delayed\_work，原来的work\_struct必须改为delayed\_work。

delayed\_work结构体定义如下：

[cpp]

```
01. struct delayed_work {
02.     struct work_struct work;
03.     struct timer_list timer;
04. };
```

为了使修改的代码最少，我做了如下修改：

1、在第56行定义了一个全局delayed\_work结构体变量 jiq\_delayed\_work;

[cpp]

```
01. static struct delayed_work jiq_delayed_work;
```

2、重新定义jiq\_read\_wq\_delayed函数如下：

[cpp]

```
01. 148static int jiq_read_wq_delayed(char *buf, char **start, off_t offset,
02. 149                             int len, int *eof, void *data)
03. 150{
04. 151     DEFINE_WAIT(wait);
05. 152     INIT_DELAYED_WORK(&jiq_delayed_work, jiq_print_wq);
06. 153
07. 154     jiq_data.len = 0;           /* nothing printed, yet */
08. 155     jiq_data.buf = buf;        /* print in this place */
09. 156     jiq_data.jiffies = jiffies; /* initial time */
10. 157     jiq_data.delay = delay;
11. 158
12. 159     prepare_to_wait(&jiq_wait, &wait, TASK_INTERRUPTIBLE);
13. 160     schedule_delayed_work(&jiq_delayed_work, delay);
14. 161     schedule();
15. 162     finish_wait(&jiq_wait, &wait);
16. 163
17. 164     *eof = 1;
18. 165     return jiq_data.len;
19. 166}
```

与原来的相比，增加了152行初始化jiq\_delayed\_work。修改了160行，调度jiq\_delayed\_work。

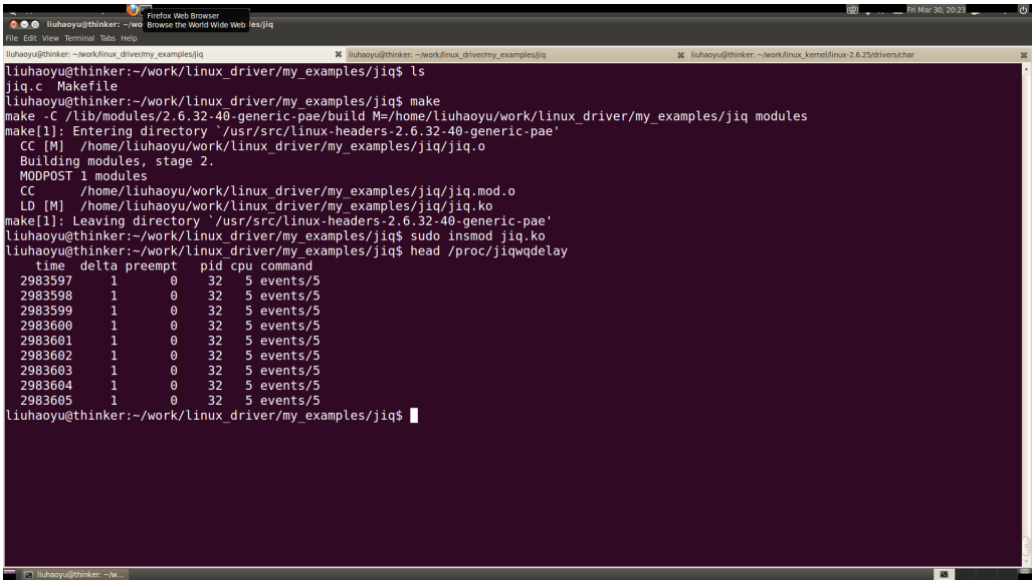
3、修改工作函数jiq\_print\_wq。

[cpp]

```
01. 113static void jiq_print_wq(struct work_struct *work)
02. 114{
03. 115     struct clientdata *data = (struct clientdata *) &jiq_data;
04. 116
05. 117     if (!jiq_print (&jiq_data))
06. 118         return;
07. 119
08. 120     if (data->delay)
09. 121         schedule_delayed_work(&jiq_delayed_work, data->delay);
10. 122     else
11. 123         schedule_work(&jiq_work);
12. 124}
```

与原来相比，修改了121行，调度jiq\_delayed\_work。

修改完成后，编译通过，如下图所示：



更多 0

上一篇 LDD3源码分析之访问控制  
下一篇 LDD3源码分析之slab高速缓存

顶 1  
踩 0

主题推荐 源码 全局变量 linux kernel concurrency structure

猜你在找

- linux驱动current, 引用当前进程, 及task\_struct
- Android 用Vibrator实现震动功能
- 国嵌视频学习——Linux内核驱动
- android中实现模拟按键
- linux下ALSA音频驱动软件开发
- SP debug info incorrect because of optimization
- struct dev\_t
- linux内核部件分析（一）——连通世界的list
- STM32 对内部FLASH读写接口函数
- 字符设备驱动程序

免费学习IT4个月,月薪12000

中国[官方授权]IT培训与就业示范基地,学成后名企直接招聘,月薪12000起!

查看评论

3楼 donghuwuwei 2014-02-14 18:45发表

C

jitc代码需要加上一个头文件<linux/sched.h>。

2楼 liuhaoyutz 2012-08-13 18:55发表

C

已经修改过来了, 谢谢提醒!

1楼 AzRael\_AreS 2012-08-13 07:55发表

朋友, 中间一大段好像乱码, 能否重新发一次

您还没有登录,请[\[登录\]](#)或[\[注册\]](#)

\* 以上用户言论只代表其个人观点, 不代表CSDN网站的观点或立场

核心技术类目

全部主题 Java VPN Android iOS ERP IE10 Eclipse CRM JavaScript Ubuntu NFC  
WAP jQuery 数据库 BI HTML5 Spring Apache Hadoop .NET API HTML SDK IIS

http://blog.csdn.net/liuhaoyutz/article/details/7412931

19/20

[Fedora](#)   [XML](#)   [LBS](#)   [Unity](#)   [Splashtop](#)   [UML](#)   [components](#)   [Windows Mobile](#)   [Rails](#)   [QEMU](#)   [KDE](#)  
[Cassandra](#)   [CloudStack](#)   [FTC](#)   [coremail](#)   [OPhone](#)   [CouchBase](#)   [云计算](#)   [iOS6](#)   [Rackspace](#)  
[Web App](#)   [SpringSide](#)   [Maemo](#)   [Compuware](#)   [大数据](#)   [aptech](#)   [Perl](#)   [Tornado](#)   [Ruby](#)   [Hibernate](#)  
[ThinkPHP](#)   [Spark](#)   [HBase](#)   [Pure](#)   [Solr](#)   [Angular](#)   [Cloud Foundry](#)   [Redis](#)   [Scala](#)   [Django](#)  
[Bootstrap](#)

[公司简介](#) | [招贤纳士](#) | [广告服务](#) | [银行汇款帐号](#) | [联系方式](#) | [版权声明](#) | [法律顾问](#) | [问题报告](#) | [合作伙伴](#) | [论坛反馈](#)  
[网站客服](#)   [杂志客服](#)   [微博客服](#)   [webmaster@csdn.net](#)   [400-600-2320](#)

京 ICP 证 070598 号

北京创新乐知信息技术有限公司 版权所有

江苏乐知网络技术有限公司 提供商务支持

Copyright © 1999-2014, CSDN.NET, All Rights Reserved 