编写属于你的第一个Linux内核模块

2014/06/26 · 开发 · 4 评论 · Linux

原文出处： LinuxVoice 译文出处：linux.cn

> 曾经多少次想要在内核游荡？曾经多少次茫然不知方向？你不要再对着它迷惘，让我们指引你走向前方……

内核编程常常看起来像是黑魔法，而在亚瑟 C 克拉克的眼中，它八成就是了。Linux内核和它的用户空间是大不相同的：抛开漫不经心，你必须小心翼翼，因为你编程中的一个bug就会影响到整个系统。浮点运算做起来可不容易，堆栈固定而狭小，而你写的代码总是异步的，因此你需要想想并发会导致什么。而除了所有这一切之外，Linux内核只是一个很大的、很复杂的C程序，它对每个人开放，任何人都去读它、学习它并改进它，而你也可以是其中之一。

学习内核编程的最简单的方式也许就是写个内核模块：一段可以动态加载进内核的代码。模块所能做的事是有限的——例如，他们不能在类似进程描述符这样的公共数据结构中增减字段（LCTT译注：可能会破坏整个内核及系统的功能）。但是，在其它方面，他们是成熟的内核级的代码，可以在需要时随时编译进内核（这样就可以摒弃所有的限制了）。完全可以在Linux源代码树以外来开发并编译一个模块（这并不奇怪，它称为树外开发），如果你只是想稍微玩玩，而并不想提交修改以包含到主线内核中去，这样的方式是很方便的。

在本教程中，我们将开发一个简单的内核模块用以创建一个**/dev/reverse**设备。写入该设备的字符串将以相反字序的方式读回（“Hello World”读成“World Hello”）。这是一个很受欢迎的程序员面试难题，当你利用自己的能力在内核级别实现这个功能时，可以使你得到一些加分。在开始前，有一句忠告：你的模块中的一个bug就会导致系统崩溃（虽然可能性不大，但还是有可能的）和数据丢失。在开始前，请确保你已经将重要数据备份，或者，采用一种更好的方式，在虚拟机中进行试验。

尽可能不要用root身份

> 默认情况下，**/dev/reverse**只有root可以使用，因此你只能使用**sudo**来运行你的测试程序。要解决该限制，可以创建一个包含以下内容的**/lib/udev/rules.d/99-reverse.rules**文件：

```
SUBSYSTEM==&quot;misc&quot;, KERNEL==&quot;reverse&quot;, MODE=&quot;0666&quot
```

> 别忘了重新插入模块。让非root用户访问设备节点往往不是一个好主意，但是在开发其间却是十分有用的。这并不是说以root身份运行二进制测试文件也不是个好主意。

模块的构造

由于大多数的Linux内核模块是用C写的（除了底层的特定于体系结构的部分），所以推荐你将你的模块以单一文件形式保存（例如，reverse.c）。我们已经把完整的源代码放在GitHub上——这里我们将看其中的一些片段。开始时，我们先要包含一些常见的文件头，并用预定义的宏来描述模块：

```c
#include <linux/init.h>
#include <linux/kernel.h>
```

```
5 MODULE_LICENSE("GPL");
6 MODULE_AUTHOR("Valentine Sinitsyn <valentine.sinitsyn@gmail.com>");
7 MODULE_DESCRIPTION("In-kernel phrase reverser");
```

这里一切都直接明了，除了**MODULE_LICENSE()**：它不仅仅是一个标记。内核坚定地支持GPL兼容代码，因此如果你把许可证设置为其它非GPL兼容的（如，“Proprietary”[专利]），某些特定的内核功能将在你的模块中不可用。

什么时候不该写内核模块

> 内核编程很有趣，但是在现实项目中写（尤其是调试）内核代码要求特定的技巧。通常来讲，在没有其它方式可以解决你的问题时，你才应该在内核级别解决它。以下情形中，可能你在用户空间中解决它更好：
>
> - 你要开发一个USB驱动 —— 请查看libusb。
> - 你要开发一个文件系统 —— 试试FUSE。
> - 你在扩展Netfilter —— 那么libnetfilter_queue对你有所帮助。
>
> 通常，内核里面代码的性能会更好，但是对于许多项目而言，这点性能丢失并不严重。

由于内核编程总是异步的，没有一个**main()**函数来让Linux顺序执行你的模块。取而代之的是，你要为各种事件提供回调函数，像这个：

C

```c
1  static int __init reverse_init(void)
2  {
3      printk(KERN_INFO "reverse device has been registered\n");
4      return 0;
5  }
6
7  static void __exit reverse_exit(void)
8  {
9      printk(KERN_INFO "reverse device has been unregistered\n");
10 }
11
12 module_init(reverse_init);
13 module_exit(reverse_exit);
```

这里，我们定义的函数被称为模块的插入和删除。只有第一个的插入函数是必要的。目前，它们只是打印消息到内核环缓冲区（可以在用户空间通过**dmesg**命令访问）；**KERN_INFO**是日志级别（注意，没有逗号）。**__init**和**__exit**是属性 —— 联结到函数（或者变量）的元数据片。属性在用户空间的C代码中是很罕见的，但是内核中却很普遍。所有标记为**__init**的，会在初始化后释放内存以供重用（还记得那条过去内核的那条“Freeing unused kernel memory...[释放未使用的内核内存.....]”信息吗？）。**__exit**表明，当代码被静态构建进内核时，该函数可以安全地优化了，不需要清理收尾。最后，**module_init()**和**module_exit()**这两个宏将**reverse_init()**和**reverse_exit()**函数设置成为我们模块的生命周期回调函数。实际的函数名称并不重要，你可以称它们为**init()**和**exit()**，或者**start()**和**stop()**，你想叫什么就叫什么吧。他们都是静态声明，你在外部模块是看不到的。事实上，内核中的任何函数都是不可见的，除非明确地被导出。然而，在内核程序员中，给你的函数加上模块名前缀是约定俗成的。

这些都是些基本概念 – 让我们来做更多有趣的事情吧。模块可以接收参数，就像这样：

C

```c
1 # modprobe foo bar=1
```

使用。我们的模块需要一个缓冲区来存储参数 —— 让我们把这大小设置为用户可配置。在**MODULE_DESCRIPTION()**下添加如下三行：

```c
static unsigned long buffer_size = 8192;
module_param(buffer_size, ulong, (S_IRUSR | S_IRGRP | S_IROTH));
MODULE_PARM_DESC(buffer_size, "Internal buffer size");
```

这儿，我们定义了一个变量来存储该值，封装成一个参数，并通过sysfs来让所有人可读。这个参数的描述（最后一行）出现在modinfo的输出中。

由于用户可以直接设置**buffer_size**，我们需要在**reverse_init()**来清除无效取值。你总该检查来自内核之外的数据 —— 如果你不这么做，你就是将自己置身于内核异常或安全漏洞之中。

```c
static int __init reverse_init()
{
    if (!buffer_size)
        return -1;
    printk(KERN_INFO
        "reverse device has been registered, buffer size is %lu bytes\n",
        buffer_size);
    return 0;
}
```

来自模块初始化函数的非0返回值意味着模块执行失败。

导航

> 但你开发模块时，Linux内核就是你所需一切的源头。然而，它相当大，你可能在查找你所要的内容时会有困难。幸运的是，在庞大的代码库面前，有许多工具使这个过程变得简单。首先，是Cscope —— 在终端中运行的一个比较经典的工具。你所要做的，就是在内核源代码的顶级目录中运行**make cscope && cscope**。Cscope和Vim以及Emacs整合得很好，因此你可以在你最喜爱的编辑器中使用它。
>
> 如果基于终端的工具不是你的最爱，那么就访问http://lxr.free-electrons.com吧。它是一个基于web的内核导航工具，即使它的功能没有Cscope来得多（例如，你不能方便地找到函数的用法），但它仍然提供了足够多的快速查询功能。

现在是时候来编译模块了。你需要你正在运行的内核版本头文件（**linux-headers**，或者等同的软件包）和**build-essential**（或者类似的包）。接下来，该创建一个标准的Makefile模板：

```c
obj-m += reverse.o
all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

现在，调用**make**来构建你的第一个模块。如果你输入的都正确，在当前目录内会找到reverse.ko文件。使用**sudo insmod reverse.ko**插入内核模块，然后运行如下命令：

```c
$ dmesg | tail -1
[ 5905.042081] reverse device has been registered, buffer size is 8192 bytes
```

恭喜了！然而，目前这一行还只是假象而已 —— 还没有设备节点呢。让我们来搞定它。

混杂设备

在Linux中，有一种特殊的字符设备类型，叫做“混杂设备”（或者简称为“misc”）。它是专为单一接入点的小型设备驱动而设计的，而这正是我们所需要的。所有混杂设备共享同一个主设备号（10），因此一个驱动(**drivers/char/misc.c**）就可以查看它们所有设备了，而这些设备用次设备号来区分。从其他意义来说，它们只是普通字符设备。

要为该设备注册一个次设备号（以及一个接入点），你需要声明**struct misc_device**，填上所有字段（注意语法），然后使用指向该结构的指针作为参数来调用**misc_register()**。为此，你也需要包含**linux/miscdevice.h**头文件：

```c
static struct miscdevice reverse_misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "reverse",
    .fops = &reverse_fops
};
static int __init reverse_init()
{
    ...
    misc_register(&reverse_misc_device);
    printk(KERN_INFO ...
}
```

这儿，我们为名为“reverse”的设备请求一个第一个可用的（动态的）次设备号；省略号表明我们之前已经见过的省略的代码。别忘了在模块卸下后注销掉该设备。

```c
static void __exit reverse_exit(void)
{
    misc_deregister(&reverse_misc_device);
    ...
}
```

‘fops’ 字段存储了一个指针，指向一个**file_operations**结构（在Linux/fs.h中声明），而这正是我们模块的接入点。**reverse_fops**定义如下：

```c
static struct file_operations reverse_fops = {
    .owner = THIS_MODULE,
    .open = reverse_open,
    ...
    .llseek = noop_llseek
};
```

另外，**reverse_fops**包含了一系列回调函数（也称之为方法），当用户空间代码打开一个设备，读写或者关闭文件描述符时，就会执行。如果你要忽略这些回调，可以指定一个明确的回调函数来替代。这就是为什么我们将**llseek**设置为**noop_llseek()**，（顾名思义）它什么都不干。这个默认实现改变了一个文件指针，而且我们现在并不需要我们的设备可以寻址（这是今天留给你们的家庭作业）。

关闭和打开

除上并不安全：如果一个用户空间应用程序泄漏了描述符（也许是故意的），它就会霸占RAM，并导致系统不可用。在现实世界中，你总得考虑到这些可能性。但在本教程中，这种方法不要紧。

我们需要一个结构函数来描述缓冲区。内核提供了许多常规的数据结构：链接列表（双联的），哈希表，树等等之类。不过，缓冲区常常从头设计。我们将调用我们的“struct buffer”：

```c
struct buffer {
    char *data, *end, *read_ptr;
    unsigned long size;
};
```

data是该缓冲区存储的一个指向字符串的指针，而**end**指向字符串结尾后的第一个字节。**read_ptr**是**read()**开始读取数据的地方。缓冲区的size是为了保证完整性而存储的 —— 目前，我们还没有使用该区域。你不能假设使用你结构体的用户会正确地初始化所有这些东西，所以最好在函数中封装缓冲区的分配和收回。它们通常命名为**buffer_alloc()**和**buffer_free()**。

static struct buffer buffer_alloc(unsigned long size) { struct buffer *buf; buf = kzalloc(sizeof(buf), GFP_KERNEL); if (unlikely(!buf)) goto out; ... out: return buf; }

内核内存使用**kmalloc()**来分配，并使用**kfree()**来释放；**kzalloc()**的风格是将内存设置为全零。不同于标准的**malloc()**，它的内核对应部分收到的标志指定了第二个参数中请求的内存类型。这里，**GFP_KERNEL**是说我们需要一个普通的内核内存（不是在DMA或高内存区中）以及如果需要的话函数可以睡眠（重新调度进程）。**sizeof(*buf)**是一种常见的方式，它用来获取可通过指针访问的结构体的大小。

你应该随时检查**kmalloc()**的返回值：访问NULL指针将导致内核异常。同时也需要注意**unlikely()**宏的使用。它（及其相对宏**likely()**）被广泛用于内核中，用于表明条件几乎总是真的（或假的）。它不会影响到控制流程，但是能帮助现代处理器通过分支预测技术来提升性能。

最后，注意**goto**语句。它们常常为认为是邪恶的，但是，Linux内核（以及一些其它系统软件）采用它们来实施集中式的函数退出。这样的结果是减少嵌套深度，使代码更具可读性，而且非常像更高级语言中的**try-catch**区块。

有了**buffer_alloc()**和**buffer_free()**，**open**和**close**方法就变得很简单了。

```c
static int reverse_open(struct inode *inode, struct file *file)
{
    int err = 0;
    file->private_data = buffer_alloc(buffer_size);
    ...
    return err;
}
```

struct file是一个标准的内核数据结构，用以存储打开的文件的信息，如当前文件位置（**file->f_pos**）、标志(**file->f_flags**）、或者打开模式（**file->f_mode**)等。另外一个字段**file->privatedata**用于关联文件到一些专有数据，它的类型是void *，而且它在文件拥有者以外，对内核不透明。我们将一个缓冲区存储在那里。

如果缓冲区分配失败，我们通过返回否定值（**-ENOMEM**）来为调用的用户空间代码标明。一个C库中调用的**open(2)**系统调用(如**glibc**)将会检测这个并适当地设置**errno**。

学习如何读和写

向地存储该字段，不需要任何额外的存储。**read**方法仅仅是从内核缓冲区复制数据到用户空间。但是如果缓冲区还没有数据，**revers_eread()**会做什么呢？在用户空间中，**read()**调用会在有可用数据前阻塞它。在内核中，你就必须等待。幸运的是，有一项机制用于处理这种情况，就是‘wait queues’。

想法很简单。如果当前进程需要等待某个事件，它的描述符（**struct task_struct**存储‘current’信息）被放进非可运行（睡眠中）状态，并添加到一个队列中。然后**schedule()**就被调用来选择另一个进程运行。生成事件的代码通过使用队列将等待进程放回**TASK_RUNNING**状态来唤醒它们。调度程序将在以后在某个地方选择它们之一。Linux有多种非可运行状态，最值得注意的是**TASK_INTERRUPTIBLE**（一个可以通过信号中断的睡眠）和**TASK_KILLABLE**（一个可被杀死的睡眠中的进程）。所有这些都应该正确处理，并等待队列为你做这些事。

一个用以存储读取等待队列头的天然场所就是结构缓冲区，所以从为它添加**wait_queue_headt read\queue**字段开始。你也应该包含**linux/sched.h**头文件。可以使用DECLARE_WAITQUEUE()宏来静态声明一个等待队列。在我们的情况下，需要动态初始化，因此添加下面这行到**buffer_alloc()**：

```c
init_waitqueue_head(&buf->read_queue);
```

我们等待可用数据；或者等待**read_ptr != end**条件成立。我们也想要让等待操作可以被中断（如，通过Ctrl+C）。因此，“read”方法应该像这样开始：

```c
static ssize_t reverse_read(struct file *file, char __user * out,
        size_t size, loff_t * off)
{
    struct buffer *buf = file->private_data;
    ssize_t result;
    while (buf->read_ptr == buf->end) {
        if (file->f_flags & O_NONBLOCK) {
            result = -EAGAIN;
            goto out;
        }
        if (wait_event_interruptible
        (buf->read_queue, buf->read_ptr != buf->end)) {
            result = -ERESTARTSYS;
            goto out;
        }
    }
...
```

我们让它循环，直到有可用数据，如果没有则使用**wait_event_interruptible()**（它是一个宏，不是函数，这就是为什么要通过值的方式给队列传递）来等待。好吧，如果**wait_event_interruptible()**被中断，它返回一个非0值，这个值代表**-ERESTARTSYS**。这段代码意味着系统调用应该重新启动。**file->f_flags**检查以非阻塞模式打开的文件数：如果没有数据，返回**-EAGAIN**。

我们不能使用**if()**来替代**while()**，因为可能有许多进程正等待数据。当**write**方法唤醒它们时，调度程序以不可预知的方式选择一个来运行，因此，在这段代码有机会执行的时候，缓冲区可能再次空出。现在，我们需要将数据从**buf->data** 复制到用户空间。**copy_to_user()**内核函数就干了此事：

```c
 size = min(size, (size_t) (buf->end - buf->read_ptr));
    if (copy_to_user(out, buf->read_ptr, size)) {
        result = -EFAULT;
        goto out;
    }
```

相信任何来自内核外的事物！

```c
    buf->read_ptr += size;
     result = size;
out:
     return result;
}
```

为了使数据在任意块可读，需要进行简单运算。该方法返回读入的字节数，或者一个错误代码。

写方法更简短。首先，我们检查缓冲区是否有足够的空间，然后我们使用**copy_from_userspace()**函数来获取数据。再然后**read_ptr**和结束指针会被重置，并且反转存储缓冲区内容：

```c
 buf->end = buf->data + size;
    buf->read_ptr = buf->data;
    if (buf->end > buf->data)
        reverse_phrase(buf->data, buf->end - 1);
```

这里， **reverse_phrase()**干了所有吃力的工作。它依赖于**reverse_word()**函数，该函数相当简短并且标记为内联。这是另外一个常见的优化；但是，你不能过度使用。因为过多的内联会导致内核映像徒然增大。

最后，我们需要唤醒**read_queue**中等待数据的进程，就跟先前讲过的那样。**wake_up_interruptible()**就是用来干此事的：

```c
wake_up_interruptible(&buf->read_queue);
```

耶！你现在已经有了一个内核模块，它至少已经编译成功了。现在，是时候来测试了。

调试内核代码

或许，内核中最常见的调试方法就是打印。如果你愿意，你可以使用普通的**printk()**（假定使用**KERN_DEBUG**日志等级）。然而，那儿还有更好的办法。如果你正在写一个设备驱动，这个设备驱动有它自己的“struct device”，可以使用**pr_debug()**或者**dev_dbg()**：它们支持动态调试（**dyndbg**）特性，并可以根据需要启用或者禁用（请查阅**Documentation/dynamic-debug-howto.txt**）。对于单纯的开发消息，使用**pr_devel()**，除非设置了DEBUG，否则什么都不会做。要为我们的模块启用DEBUG，请添加以下行到Makefile中：

```
CFLAGS_reverse.o := -DDEBUG
```

完了之后，使用**dmesg**来查看**pr_debug()**或**pr_devel()**生成的调试信息。 或者，你可以直接发送调试信息到控制台。要想这么干，你可以设置**console_loglevel**内核变量为8或者更大的值（**echo 8 /proc/sys/kernel/printk**），或者在高日志等级，如**KERN_ERR**，来临时打印要查询的调试信息。很自然，在发布代码前，你应该移除这样的调试声明。

注意内核消息出现在控制台，不要在Xterm这样的终端模拟器窗口中去查看；这也是在内核开发时，建议你不在X环境下进行的原因。

编译模块，然后加载进内核：

```
$ make
$ sudo insmod reverse.ko buffer_size=2048
$ lsmod
reverse 2419 0
$ ls -l /dev/reverse
crw-rw-rw- 1 root root 10, 58 Feb 22 15:53 /dev/reverse
```

一切似乎就位。现在，要测试模块是否正常工作，我们将写一段小程序来翻转它的第一个命令行参数。**main()**（再三检查错误）可能看上去像这样：

```
int fd = open("/dev/reverse", O_RDWR);
write(fd, argv[1], strlen(argv[1]));
read(fd, argv[1], strlen(argv[1]));
printf("Read: %s\n", argv[1]);
```

像这样运行：

```
$ ./test 'A quick brown fox jumped over the lazy dog'
Read: dog lazy the over jumped fox brown quick A
```

它工作正常！玩得更逗一点：试试传递单个单词或者单个字母的短语，空的字符串或者是非英语字符串（如果你有这样的键盘布局设置），以及其它任何东西。

现在，让我们让事情变得更好玩一点。我们将创建两个进程，它们共享一个文件描述符（及其内核缓冲区）。其中一个会持续写入字符串到设备，而另一个将读取这些字符串。在下例中，我们使用了**fork(2)**系统调用，而pthreads也很好用。我也省略打开和关闭设备的代码，并在此检查代码错误（又来了）：

```
char *phrase = "A quick brown fox jumped over the lazy dog";
if (fork())
    /* Parent is the writer */
    while (1)
        write(fd, phrase, len);
else
    /* child is the reader */
    while (1) {
        read(fd, buf, len);
        printf("Read: %s\n", buf);
}
```

你希望这个程序会输出什么呢？下面就是在我的笔记本上得到的东西：

```
Read: dog lazy the over jumped fox brown quick A
Read: A kcicq brown fox jumped over the lazy dog
Read: A kciuq nworb xor jumped fox brown quick A
Read: A kciuq nworb xor jumped fox brown quick A
...
```

一个指令。然而，内核确实无序并发的，随便就重新调度了**reverse_phrase()**函数内部某个地方运行着的写入操作的内核部分。如果在写入操作结束前就调度了**read()**操作呢？就会产生数据不完整的状态。这样的bug非常难以找到。但是，怎样来处理这个问题呢？

基本上，我们需要确保在写方法返回前没有**read**方法能被执行。如果你曾经编写过一个多线程的应用程序，你可能见过同步原语（锁），如互斥锁或者信号。Linux也有这些，但有些细微的差别。内核代码可以运行进程上下文（用户空间代码的“代表”工作，就像我们使用的方法）和终端上下文（例如，一个IRQ处理线程）。如果你已经在进程上下文中和并且你已经得到了所需的锁，你只需要简单地睡眠和重试直到成功为止。在中断上下文时你不能处于休眠状态，因此代码会在一个循环中运行直到锁可用。关联原语被称为自旋锁，但在我们的环境中，一个简单的互斥锁 —— 在特定时间内只有唯一一个进程能“占有”的对象 —— 就足够了。处于性能方面的考虑，现实的代码可能也会使用读-写信号。

锁总是保护某些数据（在我们的环境中，是一个“struct buffer”实例），而且也常常会把它们嵌入到它们所保护的结构体中。因此，我们添加一个互斥锁（‘struct mutex lock’）到“struct buffer”中。我们也必须用**mutex_init()**来初始化互斥锁；**buffer_alloc**是用来处理这件事的好地方。使用互斥锁的代码也必须包含**linux/mutex.h**。

互斥锁很像交通信号灯 —— 要是司机不看它和不听它的，它就没什么用。因此，在对缓冲区做操作并在操作完成时释放它之前，我们需要更新**reverse_read()**和**reverse_write()**来获取互斥锁。让我们来看看**read**方法 —— **write**的工作原理相同：

```c
static ssize_t reverse_read(struct file *file, char __user * out,
        size_t size, loff_t * off)
{
    struct buffer *buf = file->private_data;
    ssize_t result;
    if (mutex_lock_interruptible(&buf->lock)) {
        result = -ERESTARTSYS;
        goto out;
}
```

我们在函数一开始就获取锁。**mutex_lock_interruptible()**要么得到互斥锁然后返回，要么让进程睡眠，直到有可用的互斥锁。就像前面一样，**_interruptible**后缀意味着睡眠可以由信号来中断。

```c
while (buf->read_ptr == buf->end) {
        mutex_unlock(&buf->lock);
        /* ... wait_event_interruptible() here ... */
        if (mutex_lock_interruptible(&buf->lock)) {
            result = -ERESTARTSYS;
            goto out;
        }
    }
```

下面是我们的“等待数据”循环。当获取互斥锁时，或者发生称之为“死锁”的情境时，不应该让进程睡眠。因此，如果没有数据，我们释放互斥锁并调用**wait_event_interruptible()**。当它返回时，我们重新获取互斥锁并像往常一样继续：

```c
if (copy_to_user(out, buf->read_ptr, size)) {
        result = -EFAULT;
        goto out_unlock;
    }
    ...
out_unlock:
```

```
9     return result;
```

最后，当函数结束，或者在互斥锁被获取过程中发生错误时，互斥锁被解锁。重新编译模块（别忘了重新加载），然后再次进行测试。现在你应该没发现毁坏的数据了。

接下来是什么？

现在你已经尝试了一次内核黑客。我们刚刚为你揭开了这个话题的外衣，里面还有更多东西供你探索。我们的第一个模块有意识地写得简单一点，在从中学到的概念在更复杂的环境中也一样。并发、方法表、注册回调函数、使进程睡眠以及唤醒进程，这些都是内核黑客们耳熟能详的东西，而现在你已经看过了它们的运作。或许某天，你的内核代码也将被加入到主线Linux源代码树中 —— 如果真这样，请联系我们！

相关文章

- 自动补全不算什么，一键直达目录才是终极神器 · 1
- Linux 命令行工具使用小贴士及技巧（2）
- Linux 命令行工具使用小贴士及技巧（1）
- Linux 日志终极指南
- 每天一个 Linux 命令（60）：scp命令 · 1

可能感兴趣的话题

- 参数设置相同，词典相同，代码相同，两台电脑jieba分词结果不同？
- 2016华为校招笔试：最高分是多少 · 1
- 网易2017春招笔试：分饼干 · 1
- 你们遇到过 因为机房温度高 天天下班要关服务器的吗 哪天加班要跟助理报... · 6
- Python GUI 多个对话框 · 4
- 网易2017春招笔试：堆砖块 · 2

最新评论

Vimer 2014/06/29

不错，有机会也要自己编译一下~

赞 回复

breaker_CHN （ 1 ） 2014/07/02
用烙铁的码农+FAE

译文转发, github地址咋漏掉了? 这么重要的东西. 小编不想要奖金了吧?

赞 回复

雪中悍刀行 2014/10/04

请问如何分享或者收藏好的文章了，新来的，找了半天没找到！

赞 回复

心自知 2016/08/19

Github地址：https://github.com/vsinitsyn/reverse

赞 回复

- 本周热门文章
- 本月热门文章
- 热门标签

0 我是小有成就，但我过不了白板面试

1 趣文：程序员相亲指南

2 技术的成长曲线

3 未来是属于算法的，不是代码

4 第 0 期技术微周刊，从经典的 Lin...

5 自动补全不算什么，一键直达目录才是...

6 开发者技能修炼的五个等级

7 阿里巴巴 Java 开发手册评述

9 零 bug 策略：要么立马修复，要么...

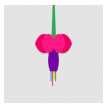
业界热点资讯 **更多 »**

又一个知名公司删了生产数据库
1 天前 · 15 · 1

2017年美国科技公司薪资排行：谷歌未进前三
21 小时前 · 3 · 2

COBOL 程序员年事已高，但依旧抢手
2 天前 · 11 · 6

GNOME 3.24首个维护版本更新发布
21 小时前 · 2



Google 神秘 Fuchsia OS 的开源线索
21 小时前 · 2

精选工具资源 **更多资源 »**

[Caffe：一个深度学习框架](#)
[机器学习](#)

[静态代码分析工具清单：公司篇](#)
[静态代码分析](#)

[HotswapAgent：支持无限次重定义运行时类与资源](#)
[开发流程增强工具](#)

[静态代码分析工具清单：开源篇（各语言）](#)
[静态代码分析](#) · 1

[Bugsnag：跨平台的错误监测](#)
[开发库](#)

关于伯乐在线博客

在这个信息爆炸的时代，人们已然被大量、快速并且简短的信息所包围。然而，我们相信：过多“快餐”式的阅读只会令人“虚胖”，缺乏实质的内涵。伯乐在线内容团队正试图以我们微薄的力量，把优秀的原创文章和译文分享给读者，为“快餐”添加一些“营养”元素。

快速链接
网站使用指南 »
问题反馈与求助 »
加入我们 »
网站积分规则 »
网站声望规则 »

关注我们

新浪微博：@伯乐在线官方微博
RSS：订阅地址

合作联系
Email：bd@Jobbole.com
QQ： 2302462408（加好友请注明来意）

更多频道

小组 – 好的话题、有启发的回复、值得信赖的圈子
头条 – 分享和发现有价值的内容与观点
相亲 – 为IT单身男女服务的征婚传播平台
资源 – 优秀的工具资源导航
翻译 – 翻译传播优秀的外文文章
文章 – 国内外的精选文章
设计 – UI,网页，交互和用户体验
iOS – 专注iOS技术分享
安卓 – 专注Android技术分享
前端 – JavaScript, HTML5, CSS
Java – 专注Java技术分享
Python – 专注Python技术分享

