Linux套接字编程5大陷阱

陷阱一：忽略返回状态

第一个陷阱很明显，但是它是缺乏经验的编程人员常常忽略的。如何你忽略函数调用的返回状态，那么你就可能注意不到它们失败了或者只是成功了一半。结果就是这个错误一直传递下去，以至于问题出现时已经很难定位问题到底出现在哪里了？

因此，不要忽略返回状态，而是要捕获并检测每一个返回值。看一下列表1中使用send函数的例子。

列表1. 忽略API调用返回状态

```
int status, sock, mode;

/* Create a new stream (TCP) socket */
sock = socket( AF_INET, SOCK_STREAM, 0 );

...

status = send( sock, buffer, buflen, MSG_DONTWAIT );

if (status == -1) {

  /* send failed */
  printf( "send failed: %s\n", strerror(errno) );

} else {

  /* send succeeded -- or did it? */

}
```

列表1展示了一个执行套接字函数send（通过一个套接字发送数据）的函数片段。在这个例子中，虽然函数的错误状态被捕获与检测，但是我们却忽略了send在非阻塞模式（通过MSG_DONTWAIT标记使能）时的特性。

send函数可能返回三类值：

如果内核将数据添加到发送队列以备发送，返回0；

如果调用过程中发送错误，返回-1（具体错误值通过errno变量标示，errno是线程本地数据）

如果只处理了部分数据，那么返回值标示最终发送了多少数据

由于send具有由MSG_DONTWAIT激活的非阻塞性特征，所以函数调用返回时可能已经发送了全部数据，或者只发送了部分甚至没有发送数据。在这个例子中，忽略返回状态可能导致不完整的数据发送以及其引起的数据丢失。

陷阱二. 对端套接字关闭

在UNIX系统中，几乎所有一切都可作为文件来处理。文件本身，文件夹，管道，外设，以及套接字都可当作文件。这种抽象意味着一系列设备类型可以使用一组相同的API来管理。

考虑函数read，它用来从文件读取以字节为最小单位的数据。read函数返回读取的字节数，发生错误时返回-1，读到文件尾时返回0。

如果你从一个文件读取数据直到文件结束（通过返回0标示），你应当关闭文件并认为大功告成。这同样适用于套接字，但语意稍有区别。如果你对一个套接字执行read并得到返回值0，那么这说明套接字的对端已经执行了close函数。这同读文件时一样，读取这个文件描述符不能得到更多的数据。

列表2. read函数正确使用方式

```
int sock, status;

sock = socket( AF_INET, SOCK_STREAM, 0 );

...

status = read( sock, buffer, buflen );

if (status > 0) {

  /* Data read from the socket */

} else if (status == -1) {

  /* Error, check errno, take action... */

} else if (status == 0) {

  /* Peer closed the socket, finish the close */
  close( sock );

  /* Further processing... */

}
```

使用write函数也可以到探测对端socket是否关闭。这种情况下，你将收到一个SIGPIPE信号（如果不响应（忽略或其他处理方式）这个信号，那么应用程序将直接退出），或者如果这个信号被阻塞了，那么write函数将返回-1并将errno设置为EPIPE。

陷阱三. 地址冲突错误

你可能使用bind函数绑定一个地址（接口加端口）到套接字。你可能在服务器端使用这个函数以限制使用哪个接口允许外部连接。你也可以在客户端使用这个函数以限制使用哪个接口发起连接。bind的常用方式是为服务器关联一个端口并使用任意地址（所有网卡上的IP地址），这样客户端可连接任一服务器接口（IP）。

使用bind函数的常见问题是试图绑定已经使用的端口。这个陷阱是说不许可将处于禁止状态的端口绑定给激活的套接字，这是由TCP套接字的TIME_WAIT状态导致的。这一状态将在套接字关闭后保持2到4分钟的时间。TIME_WAIT状态退出后，套接字将会被删除，端口就可以重新使用了。

等待TIME_WAIT状态结束并不 让人愉悦，尤其是当你正在开发套接字服务器并且需要停止它，以便修改些什么再重新启动时。幸运的是，有办法应对TIME_WAIT状态。你可以为套接字设置SO_REUSEADDR选项，以便端口可以即时重新使用。

考虑列表3中的例子。我们在绑定地址前，调用setsocketopt设置SO_REUSEADDR选项。为了激活地址重用，设置其中的整数参数为1（或者，设置其为0去使能地址重用）。

列表3. 使用SO_REUSEADDR套接字选项避免出现“地址冲突”错误

```
int sock, ret, on;
struct sockaddr_in servaddr;

/* Create a new stream (TCP) socket */
sock = socket( AF_INET, SOCK_STREAM, 0 ):

/* Enable address reuse */
on = 1;
ret = setsockopt( sock, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on) );

/* Allow connections to port 8080 from any available interface */
memset( &servaddr, 0, sizeof(servaddr) );
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl( INADDR_ANY );
servaddr.sin_port = htons( 45000 );

/* Bind to the address (interface/port) */
ret = bind( sock, (struct sockaddr *)&servaddr, sizeof(servaddr) );
```

应用SO_REUSEADDR套接字选项后，bind函数将始终许可端口的即时重用。

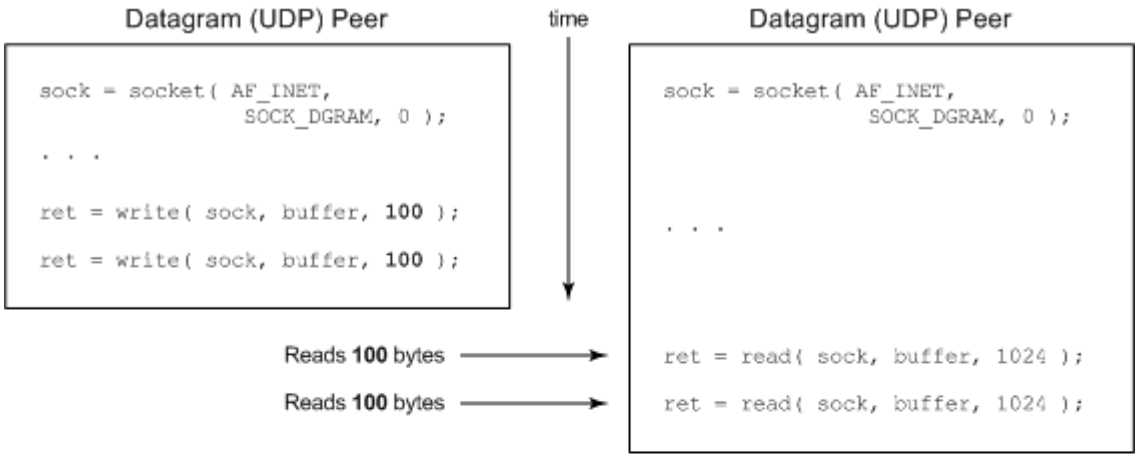
陷阱四. 发送结构化数据

这里介绍字节序转换，请参考[翻译]字节与比特序，或者直接阅读本文的原文。

陷阱五. TCP组帧假设

TCP没有提供组帧，这使得它非常适合于面向字节流的协议。这是TCP和UDP的关键不同，UDP是面向消息的协议，它将保留发送者和接收者交互的各个消息之间的界限。TCP是基于流的协议，被发送的数据将会是非结构化的。如图1所示，

图1 UDP的组帧能力对比TCP无组帧特性



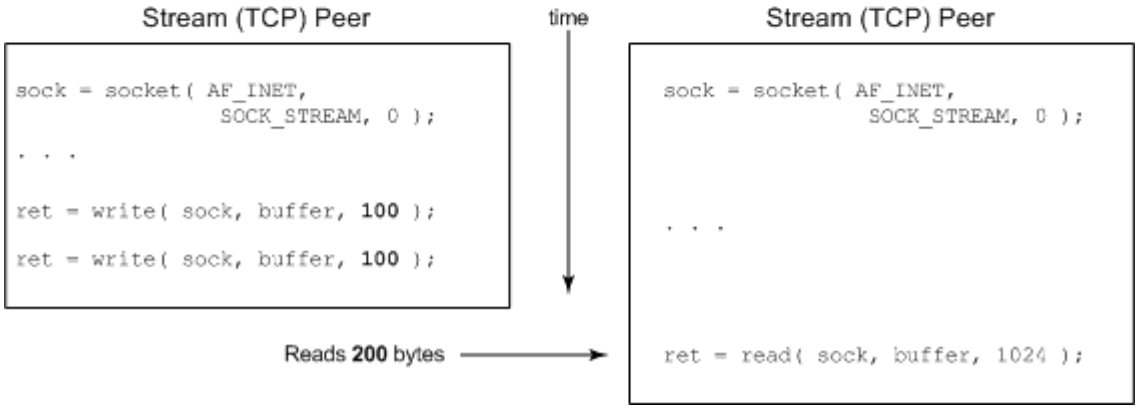


图1上给出了一个UDP客户端和服务器。左面的peer执行了两次套接字写操作，每次发送100字节。协议栈的UDP层跟踪写操作的数据量并保证右面的peer通过套接字获取数据，每次获取的数据也是100字节。换句话说，发送者给出的消息的边界也保留到了接收端。

现在看图1下。这次是两次同样的写操作，每次100字节，但是传输层是TCP（流套接字）。在这种情况下，流套接字的接收端通过一次读获取到200字节。协议栈的TCP层对两次写数据进行了组包。组包可能发生在TCP/IP协议栈的发送和接收任何一方。需要注意的是，也许根本就不会发生组包，TCP只保证数据的有序交付（有序发送给应用层，协议栈间有可能重传数据）。

这一问题让大多数开发人员不知所措。因为我们需要的是TCP的可靠性以及UDP的组帧特征。如果不能转向其他的传输层协议（如流传输控制协议（SCTP）），那么应用层开发人员需要自己实现数据缓存及分段功能。