Linux 系统中 popen 函数的缺陷与改造方案

陈志峰　王珊珊

摘　要　Linux 程序员经常会用到 popen 函数去执行 shell 命令并得到执行结果。然而 Linux 库提供的 popen 函数却有两个致命的缺陷，本文分析了这两个缺陷产生的原因，并给出改造方案。

关键词　Linux，popen，popen 函数，popen 缺陷

一、前言

熟悉 Linux 的程序员们知道可以调用系统提供的 popen 函数来执行 shell 命令并得到相关结果。然而，系统提供的 popen 函数却有着它自身的缺陷。

二、popen 函数

1. 使用方法

一个典型的 popen 函数的使用方法见程序段 1。

程序段 1：popen 函数使用

```
int main( void )
{
FILE * stream;
char buf[1024];
/ * 初始化 buf * /
memset(buf, '\0', sizeof(buf));
/ * 将" ls -l" 命令的输出 通过管道读取 (" r" 参数) 到
stream * /
stream = popen( "ls -l", "r" );
/ * 将 stream 的数据流读取到 buf 中 * /
fread(buf, sizeof(char), sizeof(buf),
stream);
/ * 以下是对 buf 内容的处理，这里略过 * /
...
pclose(stream);
return 0;
}
```

通过如上的调用，就能将 shell 命令 “ls -l” 的执行结果放到数据块 buf 中。

总结起来，popen 调用方法通常分为三步：

(1)用 popen 函数发行 shell 命令并返回一个文件描述符。

(2)对该描述符调用 fread 函数得到执行结果。

(3)用 pclose 函数将该描述符关闭。

2. 缺陷

popen 函数的调用是比较容易的。但是在实际使用中，popen 函数却有如下两个缺陷：

（1）Linux 系统的 I/O 输出分为标准输入文件（stdin），通常对应终端的键盘；标准输出文件（stdout）和标准错误输出文件（stderr），这两个文件都对应终端的屏幕。在调用 popen 函数去得到 shell 命令的结果时，只能得到标准输出文件中的内容。如果 shell 命令本身存在错误时，它会把结果（一般是一些提示信息）输出到标准错误输出文件（stderr）中，而 popen 函数不支持对标准错误输出文件内容的捕获。

（2）当发行的 shell 命令由于某种原因成为僵死进程（也就是长时间执行却不能终止的进程）时，在程序段 1 所示的代码中程序会停止在 fread 的调用处，也一直不能终止。其原因是 popen 函数内部将管道文件设置成了阻塞的性质。在调用 shell 命令时，管道文件首先被 shell 命令的进程以写的方式打开，然后在主程序中，又要以读的方式去打开。这样，当写的动作不能释放时，读的动作将一直被挂起。这种情况在某些应用中是不允许发生的。

因此，在具体应用中，如果既要求将 Linux 系统的 I/O 输出的内容全部捕获，又要避免命令发行后没有响应的情况发生，那就必须要改造这一套机制了。

3. 改造方案

popen 函数的改造方案大体上遵循了 popen 调用方法的三步，分别设计相对应的函数：

（1）popen 函数的改造

实现 popen 函数的机能主要包括如下几点：

1）能够将标准输入/输出文件和标准错误输出文件的内容捕获。

2）能够记录下 shell 命令的进程 ID。

3）能够支持可变参数 shell 命令的调用。

改造后的函数 mypopen 实现见程序段 2。

程序段 2：mypopen 的使用

```
int mypopen(const char * type, FILE * fp[2], char * execPro,
...)
{/ * mypopen 函数返回值大于 0 表明了 shell 命令的进程
ID,返回负值表明出现了错误 * /
int i,j;
/ * 这两个文件描述符捕获标准输入/输出 * /
int pfd[2];
/ * 这两个文件描述符捕获标准错误输出 * /
int pfderr[2];
pid_t pid;
int Ret;
/ * 可变参数的数据结构 * /
va_list lpArgs;
/ * OPT_MAX 表明最大可变参数的个数 * /
char * arg[OPT_MAX + 1] = {0};
/ * 标准输入/输出的管道文件的生成 * /
if(pipe(pfd) < 0) return -1;
/ * 标准输入/输出的管道文件的生成, popen 的第一个缺陷
在这里得到了解决 * /
if(pipe(pfderr) < 0){
close(pfd[0]);
close(pfd[1]);
return -1;
}
 / * 调用 fork 函数 * /
if((pid = fork()) <0){/ * fork 出现错误 * /
return -1;
}
else if(pid = = 0)
{/ * 子进程进行 shell 命令的调用 * /
if( * type = = 'r')
{/ * 从 shell 命令得到输出 * /
if(pfd[1] ! = STDOUT_FILENO)
{/ * 将标准输出复制到 pfd[1]中 * /
Ret = dup2(pfd[1], STDOUT_FILENO);
}
if(pfderr[1] ! = STDERR_FILENO)
{/ * 将错误输出复制到 pfderr[1]中 * /
Ret = dup2(pfderr[1], STDERR_FILENO);
}
}
else
{/ * 向 shell 命令输入 * /
if(pfd[0] ! = STDIN_FILENO)
{/ * 将标准输入复制到 pfd[0]中 * /
Ret = dup2(pfd[0], STDIN_FILENO);
}
if(pfderr[0] ! = STDERR_FILENO)
{/ * 将错误输出复制到 pfderr[0]中 * /
   Ret = dup2(pfderr[0], STDERR_FILENO);
}
}
/ * 可变参数的解析与 shell 的执行 * /
arg[0] = execPro;
i = 1;
va_start(lpArgs, execPro);
arg[i] = va_arg(lpArgs, char * );
while (arg[i] ! = NULL){
i + +;
if(i> = OPT_MAX) break;
arg[i] = va_arg(lpArgs, char * );
}
arg[i] = NULL;
Ret = execv(execPro, arg);
va_end(lpArgs);
_exit(0);
}
/ * 父进程将结果文件打开并返回 * /
if( * type = = 'r'){
close(pfd[1]);
close(pfderr[1]);
if((fp[0] = fdopen(pfd[0], type)) = = NULL)
{/ * 文件打开失败 * /
close(pfd[0]);
close(pfderr[0]);
return -1;
}
if((fp[1] = fdopen(pfderr[0], type)) = = NULL)
{/ * 文件打开失败 * /
fclose(fp[0]);
close(pfd[0]);
close(pfderr[0]);
return -1;
}
}
else{
close(pfd[0]);
close(pfderr[0]);
if((fp[0] = fdopen(pfd[1], type)) = = NULL)
{/ * 文件打开失败 * /
close(pfd[1]);
close(pfderr[1]);
return -1;
}
if((fp[1] = fdopen(pfderr[1], type)) = = NULL)
{/ * 文件打开失败 * /
fclose(fp[0]);
close(pfd[1]);
close(pfderr[1]);
return -1;
}
}
/ * 最后将 shell 命令的进程号作为返回值 * /
```

```
return pid;
}
```

（2）fread 调用改造

程序段 1 所示的代码中，对 popen 的执行结果是直接调用 fread 读取文件内容。前面已经说明了这种调用会产生很严重的问题。因此，为了能够确保在一个可允许的时间范围内将程序执行完毕，设置一个超时机制是必需的。而通信程序中常用的 select 模型恰好可以完成这种需求。

改造后的函数 myfread 实现见程序段 3。

程序段 3：myfread 函数的实现

```
int myfread(FILE * fp[2], void * RetVal, int RetSize, int Time-
out)
{
int size, Ret, RecvSize, MaxFD;
int Flag[2] = {0, 0};
struct timeval tv;
fd_set fdR;
/ * 首先为 select 集合选择一个合适的上限值 * /
MaxFD = max(fileno(fp[0]), fileno(fp[1]));
while (size < RetSize)
{/ * 当结果数据串的长度小于所要求的数据长度时就一直进
行读取尝试，直到 shell 命令终止 * /
/ * 下面是相关数据结构的初始化 * /
memset(& tv, 0, sizeof(tv));
tv. tv_sec = Timeout;
tv. tv_usec = 0;
FD_ZERO(& fdR);
FD_SET(fileno(fp[0]), & fdR);
FD_SET(fileno(fp[1]), & fdR);
Ret = select(MaxFD +1, & fdR, NULL, NULL, & tv);
if(Ret = = 0)
{/ * select 等待超时，做为错误返回，popen 的第二个缺陷在
这里得到了解决 * /
return -1;
}
else if (Ret > 0)
{/ * 待读取的文件中有数据了 * /
if(FD_ISSET(fileno(fp[0]), & fdR)){
RecvSize = read(fileno(fp[0]),
RetVal + size,
RetSize - size);
if (RecvSize = = 0) Flag[0] = 1;
else if (RecvSize < 0) return -1;
else size + = RecvSize;
}
if (FD_ISSET(fileno(fp[1]), & fdR))
{/ * 这里的处理和上面相似，不再列出 * /
...
}
}
else{/ * select 函数发生了错误 * /
return -1;
}
if ((Flag[0] = = 1 || Flag[1] = = 1)
& & (size < = RetSize)
{/ * 返回 0 表明 shell 命令终止了，并且我们得到了 shell 命令
的执行结果 * /
return 0;
}
}
/ * 返回 1 表明 shell 命令的输出结果还没有获取完，即超出了
RetSize 要求的长度，所以要再调用一次 myfread 继续得到输
出结果 * /
return 1;
}
```

（3）pclose 函数的改造

程序执行完成后，需要将资源释放，同样需要改造 pclose 函数。

改造后的函数 mypclose 实现见程序段 4。

程序段 4：mypclose 函数的实现

```
int mypclose(FILE * fp[2], int * processPid)
{
int states;
pid_t pid;
pid_t pid_tmp;
int ret;
/ * 先关闭文件并重置文件描述符为 NULL * /
if( fclose(fp[0]) = = EOF) return -1;
if( fclose(fp[1]) = = EOF) return -1;
fp[0] = NULL;
fp[1] = NULL;
/ * 再确保 shell 进程确实退出并重置进程 ID 为 0 * /
if((pid = * processPid) = = 0) return -1;
ret = waitpid(pid, & states, WNOHANG);
if(0 = = ret)
{/ * 如果进程仍然存在，需要手动去终止它 * /
kill(pid, SIGKILL);
do{
pid_tmp = waitpid(pid, & states, 0);
}while (pid_tmp = = -1 & & errno = = EINTR);
}
* processPid = 0;
/ * 返回进程终止状态 * /
return states;
}
```

（4）新函数集的调用方法

下面是新函数集的使用方法：

```
int main( void )
{
```

（下转第 49 页）

```
<% else %>
<% = "& nbsp; " %>
<% end if %>
</div> </td>
</tr>
<%
        r1 = rs3("hj")
        total1 = total1 + r1
        r2 = rs3("zj")
        total2 = total2 + r2
        r3 = rs3("gj")
        total3 = total3 + r3
        rs3. movenext
    next
%>
<tr>
<td colspan = "2"> <div align = "center"> <font color = "#
0000ff"> 总和 </font> </div> </td>
<td width = "46"> <div align = "center"> <font color = "#
FF0000"> <b> <% = total1 %> </b> </font> </div>
</td>
<td width = "46"> <div align = "center"> <% = total2 %>
</div> </td>
<td width = "46"> <div align = "center"> <% = total3 %>
</div> </td>
<td> & nbsp; </td>
</tr>
</table>
</html>
<%
rs1. close
rs2. close
rs3. close
conn. close
end if
%>
```

五、总结

这是多表多要求查询的一个典型例子。为了使页面源码简洁易读，在此对源码做了“瘦身”，去除了一些附加的功能代码，保留了主要的功能代码。页面中的代码，大部分读者是一看便知的语句用法。有一些是平时应用较少的语句用法，说明解释如下。

1. 根据不同的班级分别给高技班和中技班添加标识列 flag，如果读者想对这个用法加深理解，可以在 SQL Server 数据库 SQL 查询分析器中运行类似的命令，查看结果以帮助理解。

2. 有些读者可能对源码中的“join(SQL)”不太理解是什么意思，这个命令的作用是把所有的“select 1 flag，担保人，备注 from [“& trim(rs1(“班级”))& ”]union all”查询语句连接起来，最后形成用 union 合并的包含全部班级的查询语句。比如说该学校 2007 年招收了五个班级，2 个高技班，班级名称分别是：2007 高级计算机工程（1）班和 2007 高级计算机工程（2）班；3 个中技班，班级名称分别是：2007 数控技术（1）班、2007 数控技术（2）班、2001 数控技术（3）班。那“join(SQL)”的作用结果是：“select 1 flag，担保人，备注 from [2007 高级计算机工程（1）班] union all select 1 flag，担保人，备注 from 2007 高级计算机工程（2）班] union all select 0 flag，担保人，备注 from [2007 数控技术（1）班]union all select 0 flag，担保人，备注 from [2007 数控技术（2）班] union all select 0 flag，担保人，备注 from [2007 数控技术（3）班] union all”。看到这里，读者就不难明白为什么要有“left(trim(join(SQL4)), len(trim(join(SQL4))) - 9)”这样的做法了，因为在上面的结果中，最后面的 9 个字符就是 union all，所以必须去除，否则页面运行会出错。

（收稿日期：2008 年 10 月 16 日）

（上接第 35 页）

```
FILE * stream[2];
char buf[1024];
int ret, pid;
/ * 初始化 buf * /
memset(buf, '\0', sizeof(buf));
/ * 将" ls -l" 命令的输出通过管道读取（" r" 参数）到
stream * /
pid = mypopen("r", stream, "ls", " -l", NULL);
/ * 将 stream 的数据流读取到 buf 中 * /
if(pid > 0){
do{
/ * 设置 30 秒超时来调用 myfread * /
ret = myfread(stream,, buf, sizeof(buf), 30);
/ * 数据处理过程略 * /
...
if(ret < =0) break;
}while(ret = = 1) / * 当数据没读完时要再次读取 * /
/ * 资源释放 * /
mypclose(stream, & pid);
return 0;
}
```

三、结语

通过对 popen 函数的改造，得到了功能更全面、稳定性更好的一整套调用方案，从而解决了 popen 调用机制的缺陷，为制作质量更好的软件产品打下了基础。

（收稿日期：2008 年 8 月 20 日）