Expect 教程中文版
Expect 教程中文版
本文出自: 作者: 葫芦娃 翻译 (2001-09-12 10:00:00)
[版权声明]

[BUG]

有不少部分，翻译的时候不能作到“信，达”。当然了，任何时候都没有做到“雅”，希望各位谅解。

[原著]

Don Libes: National Institute of Standards and Technology
libes@cme.nist.gov

[目录]

1.[摘要]

现代的Shell对程序提供了最小限度的控制(开始，停止，等等)，而把交互的特性留给了用户。这意味着有些程序，你不能非交互的运行，比如说passwd。有一些程序可以非交互的运行，但在很大程度上丧失了灵活性，比如说fsck。这表明Unix的工具构造逻辑开始出现问题。Expect恰恰填补了其中的一些裂痕，解决了在Unix环境中长期存在着的一些问题。

Expect使用Tcl作为语言核心。不仅如此，不管程序是交互和还是非交互的，Expect都能运用。这是一个小语言和Unix的其他工具配合起来产生强大功能的经典例子。

本部分教程并不是有关Expect的实现，而是关于Expect语言本身的使用，这主要也是通过不同的脚本描述例子来体现。其中的几个例子还例证了Expect的几个新特征。

2.[关键字]

Expect,交互，POSIX,程序化的对话，Shell,Tcl,Unix;

3.[简介]

一个叫做fsck的Unix文件系统检查程序，可以从Shell里面用-y或者-n选项来执行。在手册[1]里面，-y选项的定义是象这样的。

“对于fsck的所有问题都假定一个“yes”响应；在这样使用的时候，必须特别的小心，因为它实际上允许程序无条件的继续运行，即使是遇到了一些非常严重的错误”

相比之下，-n 选项就安全的多，但它实际上几乎一点用都没有。这种接口非常的糟糕，但是却有许多的程序都是这种风格。文件传输程序 ftp 有一个选项可以禁止交互式的提问，以便能从一个脚本里面运行。但一旦发生了错误，它没有提供的处理措施。

Expect 是一个控制交互式程序的工具。他解决了 fsck 的问题，用非交互的方式实现了所有交互式的功能。Expect 不是特别为 fsck 设计的，它也能进行类似 ftp 的出错处理。

fsck 和 ftp 的问题向我们展示了象 sh,csh 和别的一些 shell 提供的用户接口的局限性。Shell 没有提供从一个程序读和象一个程序写的功能。这意味着 shell 可以运行 fsck 但只能以牺牲一部分 fsck 的灵活性做代价。有一些程序根本就不能被执行。比如说，如果没有一个用户接口交互式的提供输入，就没法运行下去。其他还有象 telnet,crypt,su,rlogin 等程序无法在 shell 脚本里面自动执行。还有很多其他的应用程序在设计是也是要求用户输入的。

Expect 被设计成专门针和交互式程序的交互。一个 Expect 程序员可以写一个脚本来描述程序和用户的对话。接着 Expect 程序可以非交互的运行“交互式”的程序。写交互式程序的脚本和写非交互式程序的脚本一样简单。Expect 还可以用于对对话的一部分进行自动化，因为程序的控制可以在键盘和脚本之间进行切换。

bes[2]里面有详细的描述。简单的说，脚本是用一种解释性语言写的(也有 C 和 C++的 Expect 库可供使用，但这超出了本文的范围)。Expect 提供了创建交互式进程和读写它们的输入和输出的命令。Expect 是由于它的一个同名的命令而命名的。

Expect 语言是基于 Tcl 的。Tcl 实际上是一个子程序库，这些子程序库可以嵌入到程序里从而提供语言服务。最终的语言有点象一个典型的 Shell 语言。里面有给变量赋值的 set 命令，控制程序执行的 if,for,continue 等命令，还能进行普通的数学和字符串操作。当然了，还可以用 exec 来调用 Unix 程序。所有这些功能，Tcl 都有。Tcl 在参考书籍 Outerhour[3][4]里有详细的描述。

Expect 是在 Tcl 基础上创建起来的，它还提供了一些 Tcl 所没有的命令。Spawn 命令激活一个 Unix 程序来进行交互式的运行。send 命令向进程发送字符串。expect 命令等待进程的某些字符串。expect 支持正规表达式并能同时等待多个字符串，并对每一个字符串执行不同的操作。expect 还能理解一些特殊情况，如超时和遇到文件尾。

expect 命令和 Tcl 的 case 命令的风格很相似，都是用一个字符串去匹配多个字符串(只要有可能，新的命令总是和已有的 Tcl 命令相似，以使得该语言保持工具族的继承性)。下面关于 expect 的定义是从手册[5]上摘录下来的。

```
expect  patlist1 action1 patlist2 action2.....
```

该命令一直等到当前进程的输出和以上的某一个模式相匹配，或者等到时间超过一个特定的时间长度，或者等到遇到了文件的结束为止。

如果最后一个 action 是空的，就可以省略它。

每一个 patlist 都由一个模式或者模式的表(lists)组成。如果有一个模式匹配成功，相应的 action 就被执行，执行的结果从 expect 返回。

被精确匹配的字符串(或者当超时发生时，已经读取但未进行匹配的字符串)被存贮在变量 expect_match 里面。如果 patlist 是 eof 或者 timeout，则发生文件结束或者超时时才执行相应的 action。一般超时的时值是 10 秒，但可以用类似"set timeout 30"之类的命令把超时时值设定为 30 秒。

下面的一个程序段是从一个有关登录的脚本里面摘取的。abort 是在脚本的别处定义的过程，而其他的 action 使用类似与 C 语言的 Tcl 原语。

```
expect "*welcome*"              break
      "*busy*"                {print busy;continue}
      "*failed*"               abort
      timeout                 abort
```

模式是通常的 C Shell 风格的正规表达式。模式必须匹配当前进程的从上一个 expect 或者 interact 开始的所有输出(所以统配符*使用的非常)的普遍。但是，一旦输出超过 2000 个字节，前面的字符就会被忘记，这可以通过设定 match_max 的值来改变。

expect 命令确实体现了 expect 语言的最好和最坏的性质。特别是，expect 命令的灵活性是以经常出

现令人迷惑的语法做代价。除了关键字模式(比如说 eof,timeout)那些模式表可以包括多个模式。这保证提供了一种方法来区分他们。但是分开这些表需要额外的扫描，如果没有恰当的用["]括起来，这有可能会把和当成空白字符。由于 Tcl 提供了两种字符串引用的方法：单引和双引，情况变的更糟。(在 Tcl 里面，如果不会出现二意性话，没有必要使用引号)。在 expect 的手册里面，还有一个独立的部分来解释这种复杂性。幸运的是：有一些很好的例子似乎阻止了这种抱怨。但是，这个复杂性很有可能在将来的版本中再度出现。为了增强可读性，在本文中，提供的脚本都假定双引号是足够的。

字符可以使用反斜杠来单独的引用，反斜杠也被用于对语句的延续，如果不加反斜杠的话，语句到一行的结尾处就结束了。这和 Tcl 也是一致的。Tcl 在发现有开的单引号或者开的双引号时都会继续扫描。而且，分号可以用于在一行中分割多个语句。这乍听起来有点让人困惑，但是，这是解释性语言的风格，但是，这确实是 Tcl 的不太漂亮的部分。

5.[callback]

令人非常惊讶的是，一些小的脚本如何的产生一些有用的功能。下面是一个拨电话号码的脚本。他用来把收费反向，以便使得长途电话对计算机计费。这个脚本用类似“expect callback.exp 12016442332”来激活。其中，脚本的名字便是 callback.exp，而+1(201)644-2332 是要拨的电话号码。

```
#first give the user some time to logout
exec sleep 4
spawn tip modem
expect "*connected*"
send "ATD [index $argv 1] "
#modem takes a while to connect
set timeout 60
expect "*CONNECT*"
```

第一行是注释，第二行展示了如何调用没有交互的 Unix 程序。sleep 4 会使程序阻塞 4 秒，以使得用户有时间来退出，因为 modem 总是会回叫用户已经使用的电话号码。

下面一行使用 spawn 命令来激活 tip 程序，以便使得 tip 的输出能够被 expect 所读取，使得 tip 能从 send 读输入。一旦 tip 说它已经连接上，modem 就会要求去拨打大哥电话号码。(假定 modem 都是贺氏兼容的，但是本脚本可以很容易的修改成能适应别的类型的 modem)。不论发生了什么，expect 都会终止。如果呼叫失败，expect 脚本可以设计成进行重试，但这里没有。如果呼叫成功，getty 会在 expect 退出后检测到 DTR，并且向用户提示 loging:。(实用的脚本往往提供更多的错误检测)。

这个脚本展示了命令行参数的使用，命令行参数存贮在一个叫做 argv 的表里面(这和 C 语言的风格很象)。在这种情况下，第一个元素就是电话号码。方括号使得被括起来的部分当作命令来执行，结果就替换被括起来的部分。这也和 C Shell 的风格很象。

这个脚本和一个大约 60K 的 C 语言程序实现的功能相似。

6.[passwd 和一致性检查]

在前面，我们提到 passwd 程序在缺乏用户交互的情况下，不能运行，passwd 会忽略 I/O 重定向，也不能嵌入到管道里边以便能从别的程序或者文件里读取输入。这个程序坚持要求真正的与用户进行交互。因为安全的原因，passwd 被设计成这样，但结果导致没有非交互式的方法来检验 passwd。这样一个对系统安全至关重要的程序竟然没有办法进行可靠的检验，真实具有讽刺意味。

passwd 以一个用户名作为参数，交互式的提示输入密码。下面的 expect 脚本以用户名和密码作为参数而非交互式的运行。

```
spawn passwd [index $argv 1]
set password [index $argv 2]
expect "*password:"
send "$password "
expect "*password:"
send "$password "
expect eof
```

第一行以用户名做参数启动 passwd 程序，为方便起见，第二行把密码存到一个变量里面。和 shell 类似，变量的使用也不需要提前声明。

在第三行，expect 搜索模式"*password:"，其中*允许匹配任意输入，所以对于避免指定所有细节而言是非常有效的。上面的程序里没有 action,所以 expect 检测到该模式后就继续运行。

一旦接收到提示后，下一行就就把密码送给当前进程，表明回车(实际上，所有的 C 的关于字符的约定都支持)。上面的程序中有两个 expect-send 序列，因为 passwd 为了对输入进行确认，要求进行两次输入。在非交互式程序里面，这是毫无必要的，但由于假定 passwd 是在和用户进行交互，所以我们的脚本还是这样做了。

最后，"expect eof"这一行的作用是在 passwd 的输出中搜索文件结束符，这一行语句还展示了关键字的匹配。另外一个关键字匹配就是 timeout 了，timeout 被用于表示所有匹配的失败而和一段特定长度的时间相匹配。在这里 eof 是非常有必要的，因为 passwd 被设计成会检查它的所有 I/O 是否都成功了，包括第二次输入密码时产生的最后一个新行。

这个脚本已经足够展示 passwd 命令的基本交互性。另外一个更加完备的例子回检查别的一些行为。比如说，下面的这个脚本就能检查 passwd 程序的别的几个方面。所有的提示都进行了检查。对垃圾输入的检查也进行了适当的处理。进程死亡，超乎寻常的慢响应，或者别的非预期的行为都进行了处理。

```
spawn passwd [index $argv 1]
expect  eof                              {exit 1}
       timeout                {exit 2}
       "*No such user.*"       {exit 3}
       "*New password:"
send  "[index $argv 2 "
expect     eof                      {exit 4}
       timeout                 {exit 2}
       "*Password too long*"     {exit 5}
       "*Password too short*"    {exit 5}
       "*Retype ew password:"
send "[index $argv 3] "
expect  timeout                  {exit 2}
       "*Mismatch*"              {exit 6}
       "*Password unchanged*"    {exit 7}
       " "
expect       timeout            {exit 2}
       "*"                     {exit 6}
       eof
```

这个脚本退出时用一个数字来表示所发生的情况。0 表示 passwd 程序正常运行，1 表示非预期的死亡，2 表示锁定，等等。使用数字是为了简单起见。expect 返回字符串和返回数字是一样简单的，即使是派生程序自身产生的消息也是一样的。实际上，典型的做法是把整个交互的过程存到一个文件里面，只有当程序的运行和预期一样的时候才把这个文件删除。否则这个 log 被留待以后进一步的检查。

这个 passwd 检查脚本被设计成由别的脚本来驱动。这第二个脚本从一个文件里面读取参数和预期的结果。对于每一个输入参数集，它调用第一个脚本并且把结果和预期的结果相比较(因为这个任务是非交互的，一个普通的老式 shell 就可以用来解释第二个脚本)。比如说，一个 passwd 的数据文件很有可能就象下面一样。

```
passwd.exp       3       bogus      -             -
passwd.exp       0       fred       abledabl       abledabl
passwd.exp       5       fred       abcdefghijklm    -
passwd.exp       5       fred       abc           -
passwd.exp       6       fred       foobar         bar
passwd.exp       4       fred       ^C            -
```

第一个域的名字是要被运行的回归脚本。第二个域是需要和结果相匹配的退出值。第三个域就是用户名。第四个域和第五个域就是提示时应该输入的密码。减号仅仅表示那里有一个域，这个域其实绝对不会用到。在第一个行中，bogus 表示用户名是非法的，因此 passwd 会响应说：没有此用户。expect 在退出时会返回 3，3 恰好就是第二个域。在最后一行中，^C 就是被切实的送给程序来验证程序是否恰当的退出。

通过这种方法，expect 可以用来检验和调试交互式软件，这恰恰是 IEEE 的 POSIX 1003.2(shell 和工具)的一致性检验所要求的。进一步的说明请参考 Libes[6]。

7.[rogue 和伪终端]

Unix 用户肯定对通过管道来和其他进程相联系的方式非常的熟悉(比如说：一个 shell 管道)。expect 使用伪终端来和派生的进程相联系。伪终端提供了终端语义以便程序认为他们正在和真正的终端进行 I/O 操作。

比如说，BSD 的探险游戏 rogue 在生模式下运行，并假定在连接的另一端是一个可寻址的字符终端。可以用 expect 编程，使得通过使用用户界面可以玩这个游戏。

rogue 这个探险游戏首先提供给你一个有各种物理属性的角色，比如说力量值。在大部分时间里，力量值都是 16，但在几乎每 20 次里面就会有一个力量值是 18。很多的 rogue 玩家都知道这一点，但没有人愿意启动程序 20 次以获得一个好的配置。下面的这个脚本就能达到这个目的。

```
for {} {1} {} {
       spawn rogue
       expect "*Str:18*"       break
            "*Str:16*"
       close
       wait
}
interact
```

第一行是个 for 循环，和 C 语言的控制格式很象。rogue 启动后，expect 就检查看力量值是 18 还是 16，如果是 16，程序就通过执行 close 和 wait 来退出。这两个命令的作用分别是关闭和伪终端的连接和等待进程退出。rogue 读到一个文件结束符就推出，从而循环继续运行，产生一个新的 rogue 游戏来检查。

当一个值为 18 的配置找到后，控制就推出循环并跳到最后一行脚本。interact 把控制转移给用户以便他们能够玩这个特定的游戏。

想象一下这个脚本的运行。你所能真正看到的就是 20 或者 30 个初始的配置在不到一秒钟的时间里掠过屏幕，最后留给你的就是一个有着很好配置的游戏。唯一比这更好的方法就是使用调试工具来玩游戏。

我们很有必要认识到这样一点：rogue 是一个使用光标的图形游戏。expect 程序员必须了解到：光标的运动并不一定以一种直观的方式在屏幕上体现。幸运的是，在我们这个例子里，这不是一个问题。将来的对 expect 的改进可能会包括一个内嵌的能支持字符图形区域的终端模拟器。

8.[ftp]

我们使用 expect 写第一个脚本并没有打印出"Hello,World"。实际上，它实现了一些更有用的功能。它能通过非交互的方式来运行 ftp。ftp 是用来在支持 TCP/IP 的网络上进行文件传输的程序。除了一些简单的功能，一般的实现都要求用户的参与。

下面这个脚本从一个主机上使用匿名 ftp 取下一个文件来。其中，主机名是第一个参数。文件名是第二个参数。

```
spawn  ftp       [index $argv 1]
expect  "*Name*"
send    "anonymous "
expect  "*Password:*"
send    [exec whoami]
expect  "*ok*ftp>*"
send    "get [index $argv 2] "
expect  "*ftp>*"
```

上面这个程序被设计成在后台进行 ftp。虽然他们在底层使用和 expect 类似的机制，但他们的可编程能力留待改进。因为 expect 提供了高级语言，你可以对它进行修改来满足你的特定需求。比如说，你可以加上以下功能：

：坚持——如果连接或者传输失败，你就可以每分钟或者每小时，甚至可以根据其他因素，比如说用户的负载，来进行不定期的重试。
：通知——传输时可以通过 mail,write 或者其他程序来通知你，甚至可以通知失败。
：初始化—每一个用户都可以有自己的用高级语言编写的初始化文件(比如说，.ftprc)。这和 C shell 对.cshrc 的使用很类似。

expect 还可以执行其他的更复杂的任务。比如说，他可以使用McGill 大学的Archie 系统。Archie 是一个匿名的Telnet 服务，它提供对描述 Internet 上可通过匿名 ftp 获取的文件的数据库的访问。通过使用这个服务，脚本可以询问Archie 某个特定的文件的位置，并把它从 ftp 服务器上取下来。这个功能的实现只要求在上面那个脚本中加上几行就可以。

现在还没有什么已知的后台-ftp 能够实现上面的几项功能，能不要说所有的功能了。在expect 里面，它的实现却是非常的简单。“坚持”的实现只要求在 expect 脚本里面加上一个循环。“通知”的实现只要执行 mail 和 write 就可以了。“初始化文件”的实现可以使用一个命令，source .ftprc，就可以了，在.ftprc 里面可以有任何的 expect 命令。

虽然这些特征可以通过在已有的程序里面加上钩子函数就可以，但这也不能保证每一个人的要求都能得到满足。唯一能够提供保证的方法就是提供一种通用的语言。一个很好的解决方法就是把 Tcl 自身融入到 ftp 和其他的程序中间去。实际上，这本来就是 Tcl 的初衷。在还没有这样做之前，expect 提供了一个能实现大部分功能但又不需要任何重写的方案。

9.[fsck]

fsck 是另外一个缺乏足够的用户接口的例子。fsck 几乎没有提供什么方法来预先的回答一些问题。你能做的就是给所有的问题都回答"yes"或者都回答"no"。

下面的程序段展示了一个脚本如何的使的自动的对某些问题回答"yes"，而对某些问题回答"no"。下面的这个脚本一开始先派生 fsck 进程，然后对其中两种类型的问题回答"yes"，而对其他的问题回答"no"。

```
for {} {1} {} {
       expect
              eof                          break
              "*UNREF FILE*CLEAR?"    {send "r "}
              "*BAD INODE*FIX?"      {send "y "}
              "*?"                  {send "n "}
}
```

在下面这个版本里面，两个问题的回答是不同的。而且，如果脚本遇到了什么它不能理解的东西，就会执行 interact 命令把控制交给用户。用户的击键直接交给 fsck 处理。当执行完后，用户可以通过按"+"键来退出或者把控制交还给 expect。如果控制是交还给脚本了，脚本就会自动的控制进程的剩余部分的运行。

```
for {} {1} {}{
       expect
              eof                          break
              "*UNREF FILE*CLEAR?"     {send "y "}
              "*BAD INODE*FIX?"       {send "y "}
              "*?"                   {interact +}
}
```

如果没有 expect，fsck 只有在牺牲一定功能的情况下才可以非交互式的运行。fsck 几乎是不可编程的，但它却是系统管理的最重要的工具。许多别的工具的用户接口也一样的不足。实际上，正是其中的一些程序的不足导致了 expect 的诞生。

10.[控制多个进程：作业控制]

expect 的作业控制概念精巧的避免了通常的实现困难。其中包括了两个问题：一个是 expect 如何处理经典的作业控制，即当你在终端上按下^Z 键时 expect 如何处理；另外一个就是 expect 是如何处理多进程的。

对第一个问题的处理是：忽略它。expect 对经典的作业控制一无所知。比如说，你派生了一个程序并且发送一个^Z 给它，它就会停下来(这是伪终端的完美之处) 而 expect 就会永远的等下去。

但是，实际上，这根本就不成一个问题。对于一个 expect 脚本，没有必要向进程发送^Z。也就是说，没有必要停下一个进程来。expect 仅仅是忽略了一个进程，而把自己的注意力转移到其他的地方。这就是 expect 的作业控制思想，这个思想也一直工作的很好。

从用户的角度来看是象这样的：当一个进程通过 spawn 命令启动时，变量 spawn_id 就被设置成某进程

的描述符。由 spawn_id 描述的进程就被认为是当前进程(这个描述符恰恰就是伪终端文件的描述符，虽然用户把它当作一个不透明的物体)。expect 和 send 命令仅仅和当前进程进行交互。所以，切换一个作业所需要做的仅仅是把该进程的描述符赋给 spawn_id。

这儿有一个例子向我们展示了如何通过作业控制来使两个 chess 进程进行交互。在派生完两个进程之后，一个进程被通知先动一步。在下面的循环里面，每一步动作都送给另外一个进程。其中，read_move 和 write_move 两个过程留给读者来实现(实际上，它们的实现非常的容易，但是，由于太长了所以没有包含在这里)。

```
spawn chess                    ;# start player one
set     id1  $spawn_id
expect      "Chess "
send           "first "             ;# force it to go first
read_move

spawn chess                     ;# start player two
set id2    $spawn_id
expect           "Chess "

for {} {1} {}{
       send_move
       read_move
       set spawn_id       $id1

       send_move
       read_move
       set spawn_id       $id2
}
```

有一些应用程序和 chess 程序不太一样，在 chess 程序里，得两个玩家轮流动。下面这个脚本实现了一个冒充程序。它能够控制一个终端以便用户能够登录和正常的工作。但是，一旦系统提示输入密码或者输入用户名的时候，expect 就开始把击键记下来，一直到用户按下回车键。这有效的收集了用户的密码和用户名，还避免了普通的冒充程序的"Incorrect password-tryagain"。而且，如果用户连接到另外一个主机上，那些额外的登录也会被记录下来。

```
spawn tip /dev/tty17          ;# open connection to
set tty $spawn_id            ;# tty to be spoofed

spawn login
set login $spawn_id

log_user 0

for {} {1} {} {
       set ready  [select $tty $login]

       case $login in $ready {
              set spawn_id $login
              expect
                 {"*password*" "*login*"}{
                       send_user $expect_match
                       set log 1
                     }
                "*"              ;# ignore everything else
              set spawn_id       $tty;
              send $expect_match
       }
       case $tty in $ready {
              set spawn_id       $tty
              expect "* *"{
                           if $log {
                             send_user $expect_match
                             set log 0
```

```
                        }
                    }
                "*"  {
                        send_user $expect_match
                    }
            set spawn_id        $login;
            send $expect_match
        }
    }
```

这个脚本是这样工作的。首先连接到一个login进程和终端。缺省的，所有的对话都记录到标准输出上(通过send_user)。因为我们对此并不感兴趣，所以，我们通过命令"log_user 0"来禁止这个功能(有很多的命令来控制可以看见或者可以记录的东西)。

在循环里面，select等待终端或者login进程上的动作，并且返回一个等待输入的spawn_id表。如果在表里面找到了一个值的话，case就执行一个action。比如说，如果字符串"login"出现在login进程的输出中，提示就会被记录到标准输出上，并且有一个标志被设置以便通知脚本开始记录用户的击键，直至用户按下了回车键。无论收到什么，都会回显到终端上，一个相应的action会在脚本的终端那一部分执行。

这些例子显示了expect的作业控制方式。通过把自己插入到对话里面，expect可以在进程之间创建复杂的I/O流。可以创建多扇出，复用扇入的，动态的数据相关的进程图。

相比之下，shell使得它自己一次一行的读取一个文件显的很困难。shell强迫用户按下控制键(比如，^C,^Z)和关键字(比如fg和bg)来实现作业的切换。这些都无法从脚本里面利用。相似的是：以非交互方式运行的shell并不处理“历史记录”和其他一些仅仅为交互式使用设计的特征。这也出现了和前面哪个passwd程序的相似问题。相似的，也无法编写能够回归的测试shell的某些动作的shell脚本。结果导致shell的这些方面无法进行彻底的测试。

如果使用expect的话，可以使用它的交互式的作业控制来驱动shell。一个派生的shell认为它是在交互的运行着，所以会正常的处理作业控制。它不仅能够解决检验处理作业控制的shell和其他一些程序的问题。还能够在必要的时候，让shell代替expect来处理作业。可以支持使用shell风格的作业控制来支持进程的运行。这意味着：首先派生一个shell，然后把命令送给shell来启动进程。如果进程被挂起，比如说，发送了一个^Z，进程就会停下来，并把控制返回给shell。对于expect而言，它还在处理同一个进程(原来那个shell)。

expect的解决方法不仅具有很大的灵活性，它还避免了重复已经存在于shell中的作业控制软件。通过使用shell，由于你可以选择你想派生的shell，所以你可以根据需要获得作业控制权。而且，一旦你需要(比如说检验的时候)，你就可以驱动一个shell来让这个shell以为它正在交互式的运行。这一点对于在检测到它们是否在交互式的运行之后会改变输出的缓冲的程序来说也是很重要的。

为了进一步的控制，在interact执行期间，expect把控制终端(是启动expect的那个终端，而不是伪终端)设置成生模式以便字符能够正确的传送给派生的进程。当expect在没有执行interact的时候，终端处于熟模式下，这时候作业控制就可以作用于expect本身。

11.[交互式的使用expect]

在前面，我们提到可以通过interact命令来交互式的使用脚本。基本上来说，interact命令提供了对对话的自由访问，但我们需要一些更精细的控制。这一点，我们也可以使用expect来达到，因为expect从标准输入中读取输入和从进程中读取输入一样的简单。但是，我们要使用expect_user和send_user来进行标准I/O，同时不改变spawn_id。

下面的这个脚本在一定的时间内从标准输入里面读取一行。这个脚本叫做timed_read，可以从csh里面调用，比如说，set answer="timed_read 30"就能调用它。

```
#!/usr/local/bin/expect -f
set timeout [index $argv 1]
expect_user "* "
send_user $expect_match
```

第三行从用户那里接收任何以新行符结束的任何一行。最后一行把它返回给标准输出。如果在特定的时间内没有得到任何键入，则返回也为空。

第一行支持"#!"的系统直接的启动脚本(如果把脚本的属性加上可执行属性则不要在脚本前面加上

expect)。当然了脚本总是可以显式的用"expect scripot"来启动。在-c后面的选项在任何脚本语句执行前就被执行。比如说，不要修改脚本本身，仅仅在命令行上加上-c "trace..."，该脚本可以加上trace功能了(省略号表示trace的选项)。

在命令行里实际上可以加上多个命令，只要中间以";"分开就可以了。比如说，下面这个命令行：

```
expect -c "set timeout 20;spawn foo;expect"
```

一旦你把超时时限设置好而且程序启动之后，expect就开始等待文件结束符或者20秒的超时时限。如果遇到了文件结束符(EOF)，该程序就会停下来，然后expect返回。如果是遇到了超时的情况，expect就返回。在这两中情况里面，都隐式的杀死了当前进程。

如果我们不使用expect而来实现以上两个例子的功能的话，我们还是可以学习到很多的东西的。在这两中情况里面，通常的解决方案都是fork另一个睡眠的子进程并且用signal通知原来的shell。如果这个过程或者读先发生的话，shell就会杀司那个睡眠的进程。传递pid和防止后台进程产生启动信息是一个让除了高手级shell程序员之外的人头痛的事情。提供一个通用的方法来象这样启动多个进程会使shell脚本非常的复杂。所以几乎可以肯定的是，程序员一般都用一个专门C程序来解决这样一个问题。

expect_user,send_user,send_error(向标准错误终端输出)在比较长的，用来把从进程来的复杂交互翻译成简单交互的expect脚本里面使用的比较频繁。在参考[7]里面，Libs描述怎样用脚本来安全的包裹(wrap)adb，怎样把系统管理员从需要掌握adb的细节里面解脱出来，同时大大的降低了由于错误的击键而导致的系统崩溃。

一个简单的例子能够让ftp自动的从一个私人的帐号里面取文件。在这种情况里，要求提供密码。即使文件的访问是受限的，你也应该避免把密码以明文的方式存储在文件里面。把密码作为脚本运行时的参数也是不合适的，因为用ps命令能看到它们。有一个解决的方法就是在脚本运行的开始调用expect_user来让用户输入以后可能使用的密码。这个密码必须只能让这个脚本知道，即使你是每个小时都要重试ftp。

即使信息是立即输入进去的，这个技巧也是非常有用。比如说，你可以写一个脚本，把你每一个主机上不同的帐号上的密码都改掉，不管他们使用的是不是同一个密码数据库。如果你要手工达到这样一个功能的话，你必须Telnet到每一个主机上，并且手工输入新的密码。而使用expect,你可以只输入密码一次而让脚本来做其它的事情。

expect_user和interact也可以在一个脚本里面混合的使用。考虑一下在调试一个程序的循环时，经过好多步之后才失败的情况。一个expect脚本可以驱动哪个调试器，设置好断点，执行该程序循环的若干步，然后将控制返回给键盘。它也可以在返回控制之前，在循环体和条件测试之间来回的切换。