

# 揭开 Expect 的神秘面纱

我们通过 Shell 可以实现简单的控制流功能，如：循环、判断等。但是对于需要交互的场合则必须通过人工来干预，有时候我们可能会需要实现和交互程序如 telnet 服务器等进行交互的功能。而 Expect 就使用来实现这种功能的工具。

Expect 是一个免费的编程工具语言，用来实现自动和交互式任务进行通信，而无需人的干预。Expect 的作者 Don Libes 在 1990 年开始编写 Expect 时对 Expect 做有如下定义：Expect 是一个用来实现自动交互功能的软件套件 (Expect [is a] software suite for automating interactive tools)。使用它系统管理员的可以创建脚本用来实现对命令或程序提供输入，而这些命令和程序是期望从终端 (terminal) 得到输入，一般来说这些输入都需要手工输入进行的。Expect 则可以根据程序的提示模拟标准输入提供给程序需要的输入来实现交互程序执行。甚至可以实现简单的 BBS 聊天机器人。

Expect 是不断发展的，随着时间的流逝，其功能越来越强大，已经成为系统管理员的一个强大助手。Expect 需要 Tcl 编程语言的支持，要在系统上运行 Expect 须首先安装 Tcl。

## Expect 工作原理

从最简单的层次来说，Expect 的工作方式象一个通用化的 Chat 脚本工具。Chat 脚本最早用于 UUCP 网络内，以用来实现计算机之间需要建立连接时进行特定的登录会话的自动化。

Chat 脚本由一系列 expect-send 对组成：expect 等待输出中输出特定的字符，通常是一个提示符，然后发送特定的响应。例如下面的 Chat 脚本实现 等待标准输出出现 Login: 字符串，然后发送 somebody 作为用户名；然后等待 Password: 提示符，并发出响应 sillyme。

```
Login: somebody Password: sillyme
```

这个脚本用来实现一个登录过程，并用特定的用户名和密码实现登录。

Expect 最简单的脚本操作模式本质上和 Chat 脚本工作模式是一样的。下面我们分析一个响应 chsh 命令的脚本。我们首先回顾一下这个交互命令的格式。假设我们为用户 chavez 改变登录脚本，命令交互过程如下：

```
# chsh chavez
Changing the login shell for chavez
Enter the new value, or press return for the default
Login Shell [/bin/bash]: /bin/tcsh
#
```

可以看到该命令首先输出若干行提示信息并且提示输入用户新的登录 shell。我们必须在提示信息后面输入用户的登录 shell 或者直接回车不修改登录 shell。

下面是一个能用来实现自动执行该命令的 Expect 脚本：

```
#!/usr/bin/expect
# Change a login shell to tcsh
set user [lindex $argv 0]
```

```
spawn chsh $user
    expect "]" : "
    send "/bin/tcsh "
    expect eof
exit
```

这个简单的脚本可以解释很多 Expect 程序的特性。和其他脚本一样首行指定用来执行该脚本的命令程序，这里是 /usr/bin/expect。程序第一行用来获得脚本的执行参数(其保存在数组 \$argv 中，从 0 号开始是参数)，并将其保存到变量 user 中。

第二个参数使用 Expect 的 spawn 命令来启动脚本和命令的会话，这里启动的是 chsh 命令，实际上命令是以衍生子进程的方式来运行的。

随后的 expect 和 send 命令用来实现交互过程。脚本首先等待输出中出现 ]: 字符串，一旦在输出中出现 chsh 输出到的特征字符串(一般特征字符串往往是等待输入的最后的提示符的特征信息)。对于其他不匹配的信息则会完全忽略。当脚本得到特征字符串时，expect 将发送 /bin/tcsh 和一个回车符给 chsh 命令。最后脚本等待命令退出(chsh 结束)，一旦接收到标识子进程已经结束的 eof 字符，expect 脚本也就退出结束。

## 决定如何响应

管理员往往有这样的需求，希望根据当前的具体情况来以不同的方式对一个命令进行响应。我们可以通过后面的例子看到 expect 可以实现非常复杂的条件响应，而仅仅通过简单的修改预处理脚本就可以实现。下面的例子是一个更复杂的 expect-send 例子：

```
expect -re "\[ (.*) ]:"
if {$expect_out(1,string) != "/bin/tcsh"} {
    send "/bin/tcsh" }
send " "
expect eof
```

在这个例子中，第一个 expect 命令现在使用了 -re 参数，这个参数表示指定的字符串是一个正则表达式，而不是一个普通的字符串。对于上面这个例子里是查找一个左方括号字符(其必须进行三次逃逸(escape)，因此有三个符号，因为它对于 expect 和正则表达时来说都是特殊字符)后面跟有零个或多个字符，最后是一个右方括号字符。这里 .\* 表示表示一个或多个任意字符，将其存放在 () 中是因为将匹配结果存放在一个变量中以实现随后的对匹配结果的访问。

当发现匹配则检查包含在 [] 中的字符串，查看是否为 /bin/tcsh。如果不是则发送 /bin/tcsh 给 chsh 命令作为输入，如果是则仅仅发送一个回车符。这个简单的针对具体情况发出不同相响应的小例子说明了 expect 的强大功能。

在一个正则表达时中，可在 () 中包含若干个部分并通过 expect\_out 数组访问它们。各个部分在表达式中从左到右进行编码，从 1 开始(0 包含有整个匹配输出)。( ) 可能会出现嵌套情况，这这种情况下编码从最内层到最外层来进行的。

## 使用超时

下一个 expect 例子中将阐述具有超时功能的提示符函数。这个脚本提示用户输入，如果在给定的时间内没有输入，则会超时并返回一个默认的响应。这个脚本接收三个参数：提示符字符串，默认响应和超时时间(秒)。

```
#!/usr/bin/expect
# Prompt function with timeout and default.
set prompt [lindex $argv 0]
set def [lindex $argv 1]
set response $def
set tout [lindex $argv 2]
```

脚本的第一部分首先是得到运行参数并将其保存到内部变量中。

```
send_tty "$prompt: "
set timeout $tout
expect " " {
    set raw $expect_out(buffer)
# remove final carriage return
    set response [string trimright "$raw" " "]
}
if {"$response" == ""} {set response $def}
send "$response "
# Prompt function with timeout and default.
set prompt [lindex $argv 0]
set def [lindex $argv 1]
set response $def
set tout [lindex $argv 2]
```

这是脚本其余的内容。可以看到 send\_tty 命令用来实现在终端上显示提示符字符串和一个冒号及空格。set timeout 命令设置后面所有的 expect 命令的等待响应的超时时间为\$tout(-1 参数用来关闭任何超时设置)。

然后 expect 命令就等待输出中出现回车字符。如果在超时之前得到回车符，那么 set 命令就会将用户输入的内容赋值给变量 raw。随后的命令将用户输入内容最后的回车符号去除以后赋值给变量 response。

然后，如果 response 中内容为空则将 response 值置为默认值(如果用户在超时以后没有输入或者用户仅仅输入了回车符)。最后 send 命令将 response 变量的值加上回车符发送给标准输出。

一个有趣的事情是该脚本没有使用 spawn 命令。该 expect 脚本会与任何调用该脚本的进程交互。

如果该脚本名为 prompt，那么它可以用在任何 C 风格的 shell 中。

```
% set a='prompt "Enter an answer" silence 10'
Enter an answer: test
```

```
% echo Answer was "$a"
Answer was test
```

prompt 设定的超时为 10 秒。如果超时或者用户仅仅输入了回车符号，echo 命令将输出 Answer was "silence"

一个更复杂的例子

下面我们将讨论一个更加复杂的 expect 脚本例子，这个脚本使用了一些更复杂的控制结构和很多复杂的交互过程。这个例子用来实现发送 write 命令给任意的用户，发送的消息来自于一个文件或者来自于键盘输入。

```
#!/usr/bin/expect
# Write to multiple users from a prepared file
# or a message input interactively
if {$argc<2} {
    send_user "usage: $argv0 file user1 user2 ... "
    exit
}
send_user 命令用来显示使用帮助信息到父进程(一般为用户的 shell)的标准输出。
set nofile 0
# get filename via the Tcl lindex function
set file [lindex $argv 0]
if {$file=="i"} {
    set nofile 1
} else {
# make sure message file exists
    if {[file isfile $file]!=1} {
        send_user "$argv0: file $file not found. "
        exit }}
}
```

这部分实现处理脚本启动参数，其必须是一个储存要发送的消息的文件名或表示使用交互输入得到发送消息的内容的“i”命令。

变量 file 被设置为脚本的第一个参数的值，是通过一个 Tcl 函数 lindex 来实现的，该函数从列表/数组得到一个特定的元素。[] 用来实现将函数 lindex 的返回值作为 set 命令的参数。

如果脚本的第一个参数是小写的“i”，那么变量 nofile 被设置为 1，否则通过调用 Tcl 的函数 isfile 来验证参数指定的文件存在，如果不存在就报错退出。

可以看到这里使用了 if 命令来实现逻辑判断功能。该命令后面直接跟判断条件，并且执行在判断条件后的 {} 内的命令。if 条件为 false 时则运行 else 后的程序块。

```
set procs {}
# start write processes
for {set i 1} {$i<$argc}
```

```

{incr i} {
spawn -noecho write
  [lindex $argv $i]
lappend procs $spawn_id
}

```

最后一部分使用 spawn 命令来启动 write 进程实现向用户发送消息。这里使用了 for 命令来实现循环控制功能，循环变量首先设置为 1，然后因此递增。循环体是最后的 {} 的内容。这里我们是用脚本的第二个和随后的参数来 spawn 一个 write 命令，并将每个参数作为发送消息的用户名。lappend 命令使用保存每个 spawn 的进程的进程 ID 号的内部变量 \$spawn\_id 在变量 procs 中构造了一个进程 ID 号列表。

```

if {$nofile==0} {
  setmesg [open "$file" "r"]
} else {
  send_user "enter message,
  ending with ^D: " }

```

最后脚本根据变量 nofile 的值实现打开消息文件或者提示用户输入要发送的消息。

```

set timeout -1
while 1 {
  if {$nofile==0} {
    if {[gets $mesg chars] == -1} break
    set line "$chars "
  } else {
    expect_user {
      -re " " {}
      eof break }
    set line $expect_out(buffer) }

    foreach spawn_id $procs {
      send $line }
    sleep 1}
exit

```

上面这段代码说明了实际的消息文本是如何通过无限循环 while 被发送的。while 循环中的 if 判断消息是如何得到的。在非交互模式下，下一行内容从消息文件中读出，当文件内容结束时 while 循环也就结束了。(break 命令实现终止循环)。

在交互模式下，expect\_user 命令从用户接收消息，当用户输入 ctrl+D 时结束输入，循环同时结束。两种情况下变量 \$line 都被用来保存下一行消息内容。当是消息文件时，回车会被附加到消息的尾部。

foreach 循环遍历 spawn 的所有进程，这些进程的 ID 号都保存在列表变量 \$procs 中，实现分别和各个进程通信。send 命令组成了 foreach 的循环体，发送一行消息到当前的 write 进程。while 循环的最后是一个 sleep 命令，主要是用于处理非交互模式情况下，以确保消息不会太快的发送给各个 write 进程。当 while 循环退出时，expect 脚本结束。

## 参考资源

Expect 软件版本本身带有很多例子脚本，不但可以用于学习和理解 expect 脚本，而且是非常使用的工具。一般可以在 /usr/doc/packages/expect/example 看到它们，在某些 linux 发布中有些 expect 脚本保存在/usr/bin 目录下。

Don Libes, Exploring Expect, O'Reilly & Associates, 1995.

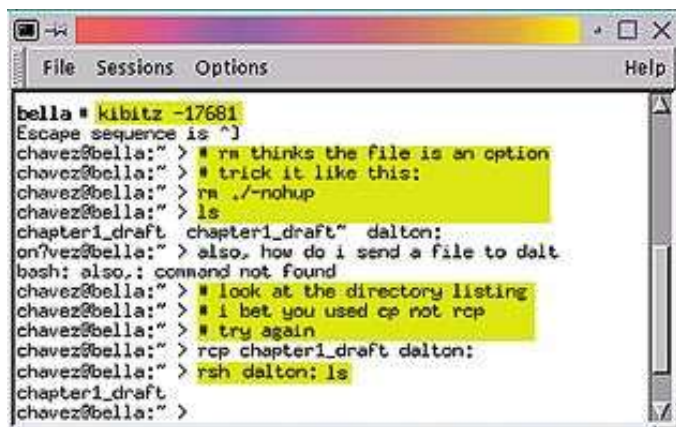
John Ousterhout, Tcl and the Tk Toolkit, Addison-Wesley, 1994.

## 一些有用的 expect 脚本

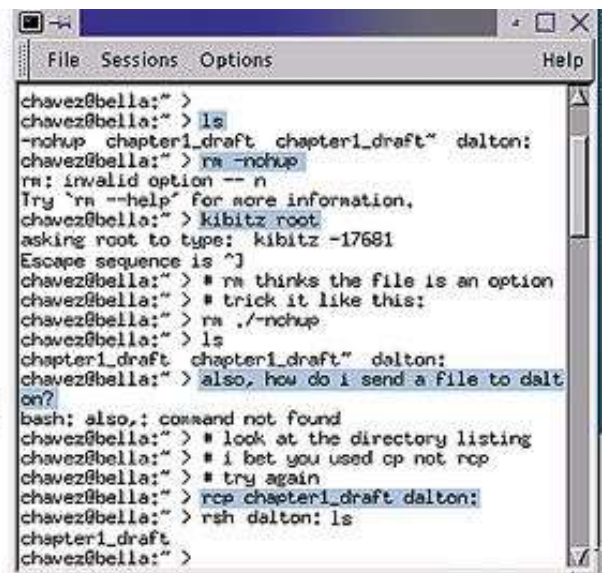
autoexpect: 这个脚本将根据自身在运行时用户的操作而生成一个 expect 脚本。它的功能某种程度上类似于在 Emacs 编辑器的键盘宏工具。一个自动创建的脚本可能是创建自己定制脚本的好的开始。

tkpasswd: 这个脚本提供了修改用户密码的 GUI 工具，包括可以检查密码是否是基于字典模式。这个工具同时是一个学习 expect 和 tk 的好实例。

Kibitz: 这是一个非常有用的工具。通过它两个或更多的用户可以连接到同一个 shell 进程。可以用于技术支持或者培训（参见图）。同样可以用于其他一些要求同步的协同任务。例如我希望和另外一个同事一起编辑一封信件，这样通过 kibitz 我们可以共享同一个运行编辑器的脚本，同时进行编辑和查看信件内容。



```
File Sessions Options Help
bella # kibitz -17681
Escape sequence is ^]
chavez@bella:~$ rm -nchup
chavez@bella:~$ ls
chapter1_draft chapter1_draft~ dalton:
chavez@bella:~$ rcp chapter1_draft dalton:
chavez@bella:~$ rsh dalton: ls
chapter1_draft
chavez@bella:~$
```



```
File Sessions Options Help
chavez@bella:~$ ls
chapter1_draft chapter1_draft~ dalton:
chavez@bella:~$ rcp chapter1_draft dalton:
chavez@bella:~$ rsh dalton: ls
chapter1_draft
chavez@bella:~$
```