Expect Examples

Updated 2016-02-14 01:24:42 by pooryorick ▲

wiki.tcl.tk 

Summary edit

Examples of Expect usage

Built-In Examples edit

Expect comes with a bunch of examples that are indispensable and unique full-function tools in their own right:

cryptdir
dislocate
kibitz
mkpasswd
multixterm
passmass
unbuffer
xkibitz
etc...

Other Examples edit

Driving tclsh with Expect
: a small beginner-level example which uses Expect to feed a script into tclsh

An example wherein Expect controls a pager
ftp-inband
Expect: Prompt Detection

The Let's Grab Everything Example edit

by Froggy

Oftentimes I just want to grab all output and do as I wish with it afterwards. Although this can be done using fileevent instead of expect (see Pipes vs Expect) it is a task well suited to demonstrate how expect grab everything from a shell command. In this example, I launch the bash shell with the [spawn] command, then I send commands to the bash shell with [exp_send], and finally I use [expect] to retrieve my results.

```
# Load the Expect package into Tcl
package require Expect

spawn bash

exp_send "ls -l\n"

set accum {}
expect {
    -regexp {..*} {
        set accum "${accum}$expect_out(0,string)"
        exp_continue
    }
}

puts $accum
```

- You must wait for this example to time out. It does not return right away. You can experiment with the -timeout option to expect. I think the default value is 10, but I am not sure.

- When I say grab everything, I mean everything. This will grab your prompt, your command, the command's results, and the prompt again, just like you see in the shell. To obtain the prompt, so that you can filter it out of your output, put an *expect* construct immediately after your spawn command. This will grab the prompt that shows up when you first launch the shell. Assign accum to some variable like "prompt".
- If the only thing you want to do with this shell is get a file listing, or if the state of the shell is already set up before you run the Tcl Script, simply use *spawn "ls -l"* or even *set processHandle [open "|ls -l 2>&1" RDONLY]*. My example is good for long-running programs like ssh, a serial session to a PLC, ftp or shell sessions where scripts that set up the environment must be run. This would be a really simple way to set up your own little GUI POP3 client (not like we're lacking those).
- **MUY IMPORTANTE**: If you want to use this inside a modularized program (imagine that), you *MUST* call *spawn* at the global level. Ie., *uplevel #0 {spawn bash}*. If you do not, *expect* will not find any results. As a matter of fact, I don't think the command issued by *exp_send* even executes. This is also true if you are using [incr tcl] RJ There are many ways to get around this. Globalizing the $spawn_id var is one, or using the -i switch with $spawn_id in exp_send works too. It is possible to spawn muliple sessions and interact with them by saving the spawn_id as you spawn new seesions then setting the spawn_id var as you need. dcd This applies also to the expect_out variable. An example

```
proc kermconnect {} {
    global opts expect_out spawn_id telnetbase

    if {$opts(host) ne ""} {
        spawn kermit -Y -j $opts(host) [expr $telnetbase + $opts(port)]
    } else {
      # use default kermit settings
        spawn kermit
    }
    set try 0
    expect {
        "C-Kermit>" { send c\r }
        timeout {
            if {$try == 0} {
                send "set prompt\r"
                incr try
                exp_continue
            }
            failed "finding C-Kermit prompt"
        }
    }
}

proc kermdisconnect {} {
    global expect_out spawn_id

    expect "Kermit>" { send "close\r"}
    expect {
        timeout {failed "close connection"}
        -re "Closing connection.*Kermit>" { send q\r; after 100 }
}
uplevel #0 catch {close}
uplevel #0 catch {wait}
```

- The `-timeout` option, if added, will be executed by [expect].
- Be sure to put `\r` after your command. If you don't, it's just like you forgot to hit enter at the keyboard.
- You don't need the "package require Expect" line if you are using the Expect executable. You need it for wish or tclsh.
- If you are not familiar with *expect_out*, it is an array. Try [array names expect_out] at the tcl shell prompt.
- `-regexp` option look strange? Check out regular expressions.

RJ Here's a simpler way to catch all of the output:

```
proc exec_it {command}     {
    spawn -noecho $command
    log_user 0
    expect eof
    return [string trimleft $expect_out(buffer) $command]
}
```

- The trimleft will just remove the echoed command from the output.

---

[1] is a SourceForge-hosted project to which Cisco contributes. Several of the Cisco-specific examples which appear there are Expect scripts.

LV: Here's a question from an expect developer:

```
#!/usr/bin/expect  --
set timeout 30
spawn /usr/local/bin/scp -P 36000 user@ip:/data/myfile  /data1
expect {
    password: {
        send "password\r"
    } "yes/no)?" {
        send "yes\r"
        set timeout -1
    } timeout {
        exit
    } eof {
        exit
    }
}
```

when scp executes normally, everything is ok. But when the host does not exist, or is not reachable, the script stops, waiting for the "password" prompt. Because the file to be copied is very large, the code needs to cancel the timeout. How would one get the result of scp so as to handle this properly? RJ Larry, made some changes to above - see if that does what you need. Also, if you only get the end of the file, try expanding the buffer with the match_max command right after the spawn.

lpenz I usually make a "match anything" mask that resets the timeout:

```
expect {
    "password:" {
        send "password\r"
    } "yes/no)?" {
        send "yes\r"
        set timeout -1
    } timeout {
        exit
    } -re . {
        exp_continue
    } eof {
        exit
    }
}
```

As scp prints the transfer progress, each char printed resets the timeout. The timeout is effectively an inactivity detector.

Capture Program Interaction edit

PYK 2016-02-13: Here is a trivial script that captures everything that appears on a terminal during the run of an interactive program. In contrast with the *nix `script` program, this doesn't capture interactive line editing, which can be a good thing.

```
#! /bin/env tclsh

spawn {*}[lrange $argv 1 end]
log_file [lindex $argv 0]
interact
```

Below is a player. redirect the output from the previous script to its stdin.

```
#! /bin/env expect

# Characters per minute
set speed 20

proc main {} {
    variable speed
    set control 10
    expect_user -re . {
        send_user -- $expect_out(buffer)
        if {$expect_out(buffer) eq "\x1b"} {
            set control 6
        } else {
            incr control -1
            incr proont
        }
        if {$control < 1 && ! [string is space $expect_out(buffer)]} {
            after [expr {60 / $speed}]
        }
        exp_continue
    } eof {}
}
main
```

Category Example	Category Expect	Category Networking