Exiting a batch file without exiting the command shell - and- batch file subroutines

devblogs.microsoft.com/oldnewthing/20120802-00

August 2, 2012

Raymond Chen

Prepare your party hats: Batch File Week is almost over.

In your batch file, you may want to exit batch file processing (say, you encountered an error and want to give up), but if you use the `exit` command, that will exit the entire command processor. Which is probably not what you intended.

Batch file processing ends when execution reaches the end of the batch file. The trick therefore is to use the `goto` command to jump to a label right before the end of the file, so that execution "falls off the end".

```
@echo off
if "%1"=="" echo You must provide a file name.&goto end
if NOT EXIST "\\server\backup\%USERNAME%\nul" mkdir "\\server\backup\%USERNAME%"
if NOT EXIST "\\server\backup\%USERNAME%\nul" echo Unable to create output
directory.&goto end
copy "%1" "\\server\backup\%USERNAME%"
:end
```

Here, there are two places where we abandon batch file execution. One is on an invalid parameter, and another is if the output directory couldn't be created (or if it isn't a directory at all).

The batch command interpreter provides a courtesy label to simply this technique: The special goto target `goto :eof` (with the colon) jumps to the end of the batch file. It's as if every batch file had a hidden goto label called `:eof` on the very last line.

The `goto :eof` trick becomes even more handy when you start playing with batch file subroutines. Okay, let's back up: Batch file subroutines?

By using the `call` command, a batch file can invoke another batch file and regain control after that other batch file returns. (If you forget the `call`, then control does not return. In other words, the default mode for batch file invocation is *chain*.) In other words, the `call`

command lets you invoke another batch file as a subroutine. The command line parameters are received by the other batch file as the usual numbered parameters `%1`, `%2`, *etc.*

It's annoying having to put every subroutine inside its own batch file, so the command interpreter folks added a way to call a subroutine *inside the same batch file*. The syntax for this is `call :label parameter parameter parameter`. This is logically equivalent to a batch file recursively calling itself, except that execution begins at the specified label instead of the first line of the file. (It's as if a secret `goto label` were added to the top of the file.)

And since it is a batch file, execution of the called subroutine ends when execution falls off the end of the file. And that's where the special `goto` target comes in handy. At the end of your subroutine, you can jump to the end of the batch file (so that execution falls off the end) by doing a `goto :eof`.

In other words, `goto :eof` is the `return` statement for batch file subroutines.

Let's take it for a spin:

```
@echo off
call :subroutine a b c
call :subroutine d e f
goto :eof
:subroutine
echo My parameters are 1=%1, 2=%2, 3=%3
goto :eof
```

That final `goto :eof` is redundant, but it's probably a good habit to get into, like putting a `break;` at the end of your last `case`.

The subroutine technique is handy even if you don't really care about the subroutine, because stashing the arguments into the `%n` parameters lets you use the tilde operators to process the inbound parameter.

```
@echo off
call :printfilesize "C:\Program Files\Windows NT\Accessories\wordpad.exe"
goto :eof
:printfilesize
echo The size of %1 is %~z1
goto :eof
```

Okay, this isn't actually much of a handy trick because you can also do it without a subroutine:

```
@echo off
for %%i ^
in ("C:\Program Files\Windows NT\Accessories\wordpad.exe") ^
do echo The size of %%i is %%~zi
```

On the other hand, the subroutine trick combines well with the `FOR` command, since it lets you put complex content in the loop body without having to mess with delayed expansion:

```
@echo off
setlocal
set DISKSIZE=1474560
set CLUSTER=512
set DISKS=1
set TOTAL=0
for %%i in (*) do call :onefile "%%i"
set /a DISKS=DISKS+1
echo Total disks required: %DISKS%
endlocal
goto :eof
:onefile
set /a SIZE=((%~z1 + CLUSTER - 1) / CLUSTER) * CLUSTER
if %SIZE% GEQ %DISKSIZE% (
    echo File %1 does not fit on a floppy - skipped
    goto :eof
)
set /a TOTAL=TOTAL+SIZE
if %TOTAL% GEQ %DISKSIZE% (
    echo ---- need another disk
    set /a DISKS=DISKS+1
    set /a TOTAL=SIZE
)
echo copy %1
goto :eof
```

This program calculates the number of floppy disks it would take to copy the contents of the current directory without compression.

The `setlocal` command takes a snapshot of the environment for restoration when we perform the `endlocal` at the end. That will clean up our temporary variables when we're done.

The first two variables are parameters for the calculation, namely the disk capacity and the cluster size. (We're assuming that the root directory can hold all the files we may ultimately copy. Hey, this is just a demonstration, not a real program.)

The next two variables are our running total of the number of disks we've used so far, and how many bytes we've used on the last disk.

The `for` command iterates over all the files in the current directory. For each one, we call `:onefile` with the file name.

The `:onefile` subroutine does all the real work. First, it takes the file size `%~z1` and rounds it up to the nearest cluster. It then sees if that size is larger than a floppy disk; if so, then we're doomed, so we just skip the file. Otherwise, we add the file to the current disk and

see if it fits. If not, then we declare the disk full and put the file on a brand new disk.

After the loop is complete, we print the number of floppy disks we calculated.

(This algorithm erroneously reports that no files require one disk. Fixing that is left as an exercise.)

There's your quick introduction to the secret `:eof` label and batch file subroutines.

[Raymond is currently away; this message was pre-recorded.]

Raymond Chen

Follow