PhUSE 2009

Paper CC01

Can I CALL EXECUTE here?

Vinod Kaippillil Mukundaprasad, Roche, Welwyn, UK

ABSTRACT

Often people find CALL EXECUTE as too sophisticated method and can be averse to use it. CALL EXECUTE is a powerful tool which serve useful solutions for many different programming scenarios and help us to come with optimal coding.

This paper will explain some of the many uses of CALL EXECUTE subroutine, presents some caveats about the use of the routine and some examples of its usage.

INTRODUCTION

CALL EXECUTE is a very powerful and useful subroutine which can be used in many different situations for many different purposes. This enables the user to write compact and efficient piece of code. The CALL EXECUTE routine allows us to build statements that are stacked for later execution. Most of the actions that can be done with a macro program can be performed by a DATA step with the use of the CALL EXECUTE statement.

CALL EXECUTE statements are resolved first and then moved to the input stack while iterating the data step. CALL EXECUTE resolves its argument and executes the resolved value at the next step boundary. If argument resolves to macro invocation, the macro executes immediately and DATA step execution pauses while the macro executes.

In this paper I will show the association between CALL EXECUTE and CALL SYMPUT in brief and some examples of the usage of CALL EXECUTE where this subroutine come up with a better alternative to the usual way.

SYNTAX

CALL EXECUTE (argument):

Argument can be one of the following:

- a character string, enclosed in quotation marks. *Argument* within single quotation marks resolves during program execution. *Argument* within double quotation marks resolves while the DATA step is being constructed. For example, to invoke the macro *mymacro*, you can use the following code:

```
call execute('%mymacro') ;
```

- the name of a DATA step character variable whose value is a text expression or a SAS statement to be generated. Do not enclose the name of the DATA step variable in quotation marks. For example, to use the value of the DATA step variable COUNT, which contains a SAS statement or text expression, you can use the following code:

```
call execute(count) ;
```

- a character expression that is resolved by the DATA step to a macro text expression or a SAS statement. For example, to generate a macro invocation whose parameter is the value of the variable COUNT, you use the following code:

```
call execute('%mymacro'||count||')') ;
```

CALL SYMPUT AND EXECUTE

The CALL SYMPUT allows the data step to add a new macro variable or overwrite an existing one to the macro space. This action is not performed until the data step is completed its processing. So it is a common knowledge that the macro variable created by CALL SYMPUT cannot be referenced inside the creating data step and you have to use another step outside this data step if you want to check the value of this macro variable or to use it. But CALL EXECUTE gives you the flexibility to reference the macro variable inside the data step used to create it. See the example below. This is possible because any SAS statements passed through CALL EXECUTE are executed after the data step finished its execution.

```
data _null_ ;
   call symput (‘myobs’, 100) ;
   call execute(“%put The macro variable created here is &myobs; “) ;
run ;
```

Though we have the above mentioned advantage in the usage of CALL EXECUTE with CALL SYMPUT, we should be careful in the usage of this subroutine. One limitation of CALL EXECUTE lies with the usage of CALL SYMPUT. The EXECUTE subroutine cannot be used to call a macro that has macro variables created by CALL SYMPUT and used that macro variable inside the same macro. The following code will generate warning in the SAS log about the unresolved macro variable as shown below.

SAS Code:

```
%macro test ;
   data _null_ ;
      call symput(‘mvar’, 100) ;
   run ;
   %put Value of mvar is &mvar ;
%mend ;

data _null_ ;
   call execute( "%test") ;
run ;
```

SAS Log:

```
1    %macro test ;
2       data _null_ ;
3          call symput('mvar', 100) ;
4       run ;
5       %put Value of mvar is &mvar ;
6    %mend ;
7
8    data _null_ ;
9       call execute( "%test") ;
WARNING: Apparent symbolic reference MVAR not resolved.
Value of mvar is &mvar
10   run ;

NOTE: DATA statement used:
      real time           0.01 seconds
      cpu time            0.01 seconds


NOTE: CALL EXECUTE generated line.
1   + data _null_ ;       call symput('mvar', 100) ;    run ;

NOTE: Numeric values have been converted to character values at the places given by:
(Line):(Column).
      1:42
NOTE: DATA statement used:
      real time           0.01 seconds
      cpu time            0.01 seconds
```

This is actually to do with the timing of macro references and SAS statements. It is important to know that macro references within CALL EXECUTE are executed immediately, but SAS language statements within CALL EXECUTE do not execute until after the data step containing CALL EXECUTE is executed. SAS statements generated by macros also execute after the data step has executed. That is why you cannot use CALL EXECUTE to invoke a macro that contains references to macro variables created by CALL SYMPUT in that macro.

SOME EXAMPLES

EXAMPLE 1

Suppose we need a sample print from all the data sets in WORK library. Of course there are many different ways to generate this. But you will notice the efficiency and compactness of the code using CALL EXECUTE below.

```
data _null_ ;
   set sashelp.vtable ;
   where libname = 'WORK' ;
   call execute
      ("proc print data = WORK."|| memname|| "(obs=5); run;" ) ;
run ;
```

EXAMPLE 2

Suppose we have to rename hundreds of variables beginning with "RM_" to remove "RM_". We might go to the route of PROC DATASETS to do the renaming but we do not want to call the proc hundreds of times. CALL EXECUTE provide the best option to do this.

```
proc contents data = inlib.dset out = dsout ;
run;

data _null_ ;
   if _n_ = 1 then
      call execute ("proc datasets lib = inlib; modify dset; rename ") ;
   if eof then
      call execute ("; run; quit;") ;
   set dsout end = eof ;
   if upcase(name) =: "RM_" then
      call execute( name || " = %" || "substr(" || name || ",4)") ;
run ;
```

EXAMPLE 3

In many situations we needed to use the same procedures on different datasets, mostly within same library. Usually we do that using many different proc steps or by creating a macro for the procedure and calling that as many times as we need. CALL EXECUTE enable us to do this within a single data step as below.
Consider we need to print a selected number of data sets in a library.

```
data _null_ ;
   length dsets $20 ;
   do dsets = 'dset1', 'dset2', 'dset3', 'dset4' ;
      call execute ("proc print data = inlib."
                     || trim(left(dsets)) ||  "; run ;")  ;
   end ;
run ;
```

LIMITATIONS/CONSIDERATIONS

As mentioned before, because macro references execute immediately and SAS statements do not execute until after a step boundary, you cannot use CALL EXECUTE to invoke a macro that contains references for macro variables that are created by CALL SYMPUT in that macro.

A documented problem exists in SAS release 8.2 is when using CALL EXECUTE to build large amounts of code (generally more than a thousand lines of generated code) as this may cause the code that is generated to be incomplete or truncated, although the MPRINT code looks fine. In other cases, generated code may be inappropriately repeated. We may receive a variety of errors depending on where the code was misinterpreted. A Technical Support hot fix TSLEVEL TS2M0 for this issue is available. The problem is resolved in SAS release 9.

Though the routine allows us to submit partial SAS statements to the input stack within one CALL EXECUTE statement, we should be careful when we are passing comments using CALL EXECUTE. When we use the PL/I style comments (/* */), it must be in the same CALL EXECUTE statement. The reason for this behavior is the PL/I-style comments are considered a single token, and therefore must not be split. For example, the following SAS code will not give you the expected result.

SAS Code:

```
data _null_ ;
   call execute("data test;") ;
   call execute("/* Start comment") ;
   call execute(" comment continues */") ;
   call execute("x=1; output; run; proc") ;
   call execute(" print; run;") ;
run ;
```

SAS Log:

```
1    data _null_ ;
2       call execute("data test;") ;
3       call execute("/* Start comment") ;
4       call execute(" comment continues */") ;
5       call execute("x=1; output; run; proc") ;
6       call execute(" print; run;") ;
7    run ;

NOTE: DATA statement used:
      real time           0.01 seconds
      cpu time            0.00 seconds


NOTE: CALL EXECUTE generated line.
1   + data test;
2   +  comment continues */
3   + x=1; output; run;

NOTE: The data set WORK.TEST has 1 observations and 0 variables.
NOTE: DATA statement used:
      real time           0.01 seconds
      cpu time            0.00 seconds

3   +                  proc

4   +  print; run;

NOTE: No variables in data set WORK.TEST.
NOTE: PROCEDURE PRINT used:
      real time           0.00 seconds
      cpu time            0.00 seconds
```

CONCLUSION

CALL EXECUTE helps us to combine data steps and proc together and it eliminates the need of typing same piece of code again and again. There are many different scenarios where CALL EXECUTE come to rescue. I covered only a very few cases in this paper. The CALL EXECUTE subroutine is a valuable addition to the SAS language, but the user must really understand how SAS code and macro code are processed in order to use it efficiently to the full extend and avoid any pitfalls.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Vinod Kaippillil Mukundaprasad
Roche Products
Welwyn Garden City
AL7 1TW
vinod.kaippillil_mukundaprasad@roche.com