source: http://www.MSSQLTips.com/tip.asp?id=3537 -- printed: 3/4/2015 9:11:54 PM

Executing scripts on multiple servers by SQL Server version and edition using PowerShell

Written By: Tim Smith -- 3/4/2015

Problem

As a SQL Server Professional, I maintain different SQL Server environments by version and was curious if you had any useful approaches to automating and obtaining scripts depending on the version of SQL Server.

Solution

In PowerShell, we can obtain the version of SQL Server and execute a script based on the version, which if we're looping through hundreds of servers, which may differ in version, can be helpful without re-adjusting pointers.

In general, I code with the most effective principle, meaning that I won't use new functionality if old functionality works in a newer edition provided that (1) no difference in performance exists, and (2) the old code isn't deprecated, or won't be deprecated in a future edition of SQL Server. An example of this in action would be DENSE_RANK() or ROW_NUMBER() vs. the LAG and LEAD function in certain comparative analysis; while LAG and LEAD certainly work, they don't work with every edition of SQL Server, thus DENSE_RANK() and ROW_NUMBER() have the advantage, if the performance is the same or similar.

Using the below function, I receive a script based on the server, version and name of the script. While I've kept in the location, SQL Server Professionals should note that a standard location can be hard-coded into the script, much like the SMO library is (both can also be changed as parameters).

This script assumes that code is organized by folder, where a T-SQL script for 2000 is saved in a folder titled "2000" and the same with other versions (2008 would be in the 2008 folder). I will pass in the server name ($server), the location for the scripts (such as "C:\GitHub\MSSQL\", which can be hardcoded if this is always the same), and the name of the function ($name).

```
Function Get-ScriptByVersion {
    Param (
        [string]$server
        , [string]$location
        , [string]$name
    )
    Process
    {
        $smolibrary = "C:\Program Files (x86)\Microsoft SQL Server\110\SDK\Assemblies\Microsoft.SqlServer.Smo.dll"
        Add-Type -Path $smolibrary
        $srv = New-Object Microsoft.SqlServer.Management.SMO.Server($server)
        $version = $srv.VersionMajor.ToString()
        switch ($version)
        {
            8 { $version = "2000" }
            9 { $version = "2005" }
            10 { $version = "2008" }
            11 { $version = "2012" }
            12 { $version = "2014" }
        }
        $scriptloc = $location + $version + "\" + $name + ".sql"
        $sqlscript = Get-Content $scriptloc
        return $sqlscript
    }
}
```

One useful feature that we can add to this function, especially if our development, QA and production servers differ in editions (standard, developer, enterprise) is to get the edition as follows:

```
$srv.EngineEdition
```

And then we can wrap a switch with the result of the engine edition; this would mean that our code is also separated by the edition. One alternative is if we want to add a folder for any T-SQL script that works in any edition of SQL Server; because I use the testing method, even interchangeable scripts are kept by folder, as that's a confirmation it's been tested.

Using this function, I can return the full script, which some developers, using PowerShell ISE may prefer. Below, I show how I can use a few other scripts to actually call the function against a server and database, so if it involves building procedures, tables, views, functions, etc., it will actually build it on the server and database I pass to it; this may be a preferred method for developers.

The below two functions involve taking the code extracted from the script **Get-ScriptByVersion** and running the code against the server and database passed to it. Since the **Execute-SQL** script runs the T-SQL code, it is the **Build-VersionName** that will be called. For instance, **Build-VersionName - $server "OURSERVER\OURINSTANCE" -location "N:\GitHub\MSSQL\" -name "checkdb" -database "OurNewDB"**.

```
###  The below function executes our script
Function Execute-SQL {
    Param (
    [string]$server
    , [string]$database
    , [string]$commandtext
    )
    Process
    {
        $scon = New-Object System.Data.SqlClient.SqlConnection
        $scon.ConnectionString = "Data Source=$server;Initial Catalog=$database;Integrated Security=true;Connectio
        $cmd = New-Object System.Data.SqlClient.SqlCommand
        $cmd.Connection = $scon
        $cmd.CommandText = $commandtext
        $cmd.CommandTimeout = 0
        try
  {
   $scon.Open()
   $cmd.ExecuteNonQuery() | Out-Null
  }
  catch
  {
   Write-Host "Exeption message."
  }
  finally
  {
   $scon.Dispose()
   $cmd.Dispose()
  }
    }
}


###  The below function is our wrapper function; it will call both other functions
###  We will give it all the parameters that the other two functions need: the server name, the script location, t
Function Build-VersionName {
    Param (
        [string]$server
        , [string]$location
        , [string]$name
        , [string]$database
    )
    Process
    {
        $commandscript = Get-ScriptByVersion -server $server -location $location -name $name
        Execute-SQL -server $server -database $database -commandtext $commandscript
    }
}
```

Where these functions work best is when walking into a new environment and either getting information from the servers, or building new structures on the server. If I keep the same build process, for instance, building a maintenance database on all 100 servers, I can then edit the scripts to place some direct code into them, such as the location - that may not change at all for a specific environment. But attaching a drive and launching this

script can simplify this process; in addition, if I need to query, I can output the information on ISE using **Get-ScriptByVersion**, whereas if I need to build, I can call the **Build-VersionName**.

Next Steps

- One quick check you'll want to make is ensuring your T-SQL is named the same; for instance, a 2012 procedure involving CHECKDB is named the same as a 2005 procedure involving CHECKDB; this way I can use LAG/LEAD in 2012 and newer, and ROW_NUMBER() or DENSE_RANK() in older versions, as an example.
- Test using attached and mobile drives.
- Read these other PowerShell Scripts for SQL Server