PowerShell Jobs Week: Remote Jobs

Dr Scripto

March 4th, 2014

Summary: Richard Siddaway introduces you to Windows PowerShell jobs on remote machines.

Honorary Scripting Guy, Richard Siddaway, here today filling in for my good friend, The Scripting Guy. This is the third in a series of posts that, hopefully, will show the spotlight on Windows PowerShell jobs, remind people of their capabilities, and encourage their greater adoption. The full series comprises:

1. Introduction to PowerShell Jobs
2. WMI and CIM Jobs
3. Remote Jobs (this post)
4. Scheduled Jobs
5. Jobs and Workflows
6. Job Processes
7. Jobs in the Enterprise

So far, you've seen how to use the core **Job** cmdlets and the **–AsJob** parameter, and we discovered the interesting aspects of the CIM cmdlets and CDXML. One of biggest selling points for Windows PowerShell is the way it enables you to administer remote machines. You have a number of options:

- Cmdlets with the –ComputerName parameter
- Invoke-Command
- Interactive remote sessions
- Windows PowerShell remoting sessions
- Workflows
- CIM sessions
- WSMAN cmdlets
- Desired State Configuration

All of these options enable you to perform at least some amount of administration on the remote machine. But how do Windows PowerShell jobs fit into this framework? **Start-Job** doesn't have a **–ComputerName** parameter. A couple of techniques immediately come to mind. Is there any difference, for instance, between the following two commands?

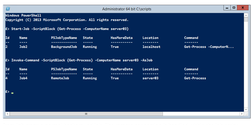
```
Start-Job -ScriptBlock {Get-Process -ComputerName server03}

Invoke-Command -ScriptBlock {Get-Process} -ComputerName server03 -AsJob
```

And what do they actually do?

They both start a job. They both call **Get-Process**. And they both run against a remote machine called server03.

In reality, though, there is a difference. The difference is about where the job runs and how you get to the data.

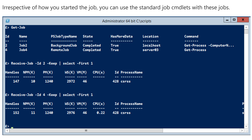


Start-Job starts a normal background job. The remote connection is handled within the job script block. The script block runs as a separate Windows PowerShell process (more on that in Part 6 of the series). The remote connection is managed by **Get-Process**, so it is destroyed when the data is returned. Notice that the location says **localhost**, which means that the job is running locally.

In the second job, you're using **Invoke-Command** to run the command and control the connection to the remote machine. **Invoke-Command** uses standard Windows PowerShell remoting for connectivity to remote machines. The **–AsJob** parameter brings the Windows PowerShell job engine into play. Notice that this time the PSJobTypeName is **RemoteJob** and the location is server03 (the remote machine).

The big difference is where the job runs. By using **Start-Job**, it runs locally and the code in the script block handles the connectivity. By using **Invoke-Command**, the job runs on the remote machine, but the results come back to the local machine.

Irrespective of how you started the job, you can use the standard job cmdlets with these jobs.



Get-Job works as you would expect. You can use **Receive-Job** in exactly the same way on both jobs.

There are a number of broad scenarios for running jobs on remote machines:

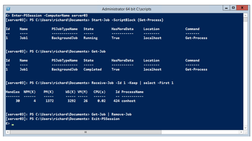
Option 1: Run a normal job and perform the remote connectivity inside the job's script block. This is demonstrated in the first of the examples you've seen.

Option 2: Start an interactive session to a remote server. If you start a job in the interactive session, you have exactly the same experience as if you were running on the local machine, but all actions are performed on the remote machine.

Option 3: Run the background job on a remote machine and have the data returned to the local machine. This is demonstrated by the second of the examples you've seen.

Option 4: Run the background job on the remote machine and keep the results on the remote machine. This very similar to Option 2, but you are using a Windows PowerShell remote session rather than an interactive session.

You've seen examples of Options 1 and 3 (although, I'll provide another example later that uses a remote session). So let's look at running jobs in an interactive session.



Interactive remote sessions have to be one of the coolest Windows PowerShell features ever. Use **Enter-PSSession** to open the session. If you already have a session open referenced by the variable **$sess**, you can enter that session:

```
$sess = New-PSSession –ComputerName server03

Enter-PSSession -Session $sess
```

When you enter the session, your prompt changes to include the remote machine name. In this case changing from

```
E:>
```

to

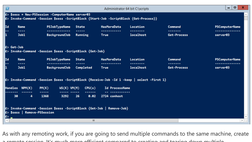
```
[server03]: PS C:\Users\richard\Documents>
```

You can then use **Start-Job** and the other job cmdlets as you would on a local machine. Notice that the location is **localhost** and the job type is **BackgroundJob**. Also notice that the first process returned is different when compared to the previous examples. This is because of the interactive session.

Each time you create a new interactive session, the job ID counters reset. Notice that the job starts with an ID of 1 rather than the first job having an ID of 2 when running locally.

When you exit the session, it is destroyed if you created it with **Enter-PSSession**. If you are using an existing session, it will remain available for use until you explicitly destroy it.

The other option that you haven't see yet is Option 4—running the job on the remote machine and keeping the data on the remote machine.



As with any remoting work, if you are going to send multiple commands to the same machine, create a remote session. It's much more efficient compared to creating and tearing down multiple connections. You also need the continuity of the session to ensure that you can access the job when it has completed.

```
$sess = New-PSSession -ComputerName server03
```

With the session in place, you can use it with **Invoke-Command** to send the **Start-Job** command to the remote machine.

```
Invoke-Command -Session $sess -ScriptBlock {Start-Job -ScriptBlock {Get-Process}}
```

You will get back the job start information, which may be confusing because the job is supposed to be running on the remote machine. Remember that **Invoke-Command** always returns any response from an issued command.

You can easily test the state of any jobs on your local machine by using:

```
Get-Job
```

In this example there were no local jobs.

Next, test that the job has completed:

```
Invoke-Command -Session $sess -ScriptBlock {Get-Job}
```

You may need to repeat the test a number of times if it's a long running job. After the job has completed, you can retrieve the data:

```
Invoke-Command -Session $sess -ScriptBlock {Receive-Job -Id 1 -keep | select -First 1}
```

At the end of your work, always clean up and remove unwanted jobs and the remote sessions.

```
Invoke-Command -Session $sess -ScriptBlock {Get-Job | Remove-Job}

$sess | Remove-PSSession
```

You were promised a look at using **Invoke-Command –AsJob** (Option 3) through a Windows PowerShell remoting session.

Create the session and use the session with **Invoke-Command** and the **–AsJob** parameter. The job will run on the remote machine, but the results will be returned to the local machine as shown in the previous image. Access the results and manage the job as any other job on your local machine.

So, one question remains…

Should we bring back the data to the local machine or keep the data on the remote machine?

That's Options 1-3 versus Option 4. As usual, the answer is, "That it depends."

If you are performing some kind of fan-out administration task, you want the data and information about job success or failure brought back to the local machine. If you can't do this, you're making more work for yourself in collecting the data.

I'd reserve the option for keeping the data on the remote machine for situations where the data may be confidential, and it would be more secure if it was left on the remote machine until you need it.

That's it for today. Tomorrow you'll learn about creating and using scheduled jobs on remote computers. Bye for now.

–Richard

Thanks again, Richard!

I invite you to follow me on Twitter and Facebook. If you have any questions, send email to me at scripter@microsoft.com, or post your questions on the Official Scripting Guys Forum. See you tomorrow. Until then, peace.

Ed Wilson, Microsoft Scripting Guy

Dr Scripto

Scripter, PowerShell, vbScript, BAT, CMD

Follow Dr Scripto

Posted in Uncategorized Tagged background jobs, Richard Siddaway, scripting techniques, Windows PowerShell

0 comments

PowerShell Resources

PowerShell Documentation
Scripting Forums
PowerShell Forums
PowerShell on TechCommunity
PowerShell.org – Community Resource

Topics

c# | .NET | .NET Core | .NET Framework | 2009 Summer Scripting Games | 2010 Scripting Games | 2011 Scripting Games | 2012 Scripting Games | 2013 Scripting Games | 2014 Scripting Games | 2014 Winter Scripting Games | 2015 Holiday Series | 4.0 | 70-410 | Aaron Nelson | Access Application | ACLs | activation | Active Directory | Active Directory Application Mode (ADAM)

Archive

October 2019
September 2019
August 2019
July 2019
September 2018
July 2018
June 2018
May 2018
March 2018
February 2018
December 2017