Gestion de logs sous PowerShell à l'aide du Framework Log4Net.

Par Laurent Dardenne, le 27 janvier 2009.

Niveau

Débutant Avancé Confimé

La journalisation d'évènements relatifs à l'exécution d'un script est assez sommaire sous PowerShell, chaque scripteur devant coder sa propre gestion de log.

La communauté .NET met volontiers à profit le travail de la communauté JAVA, par exemple en migrant le Framework Log4J sous le nom de Log4NET. Ce Framework permet la création d'évènements hiérarchisés pouvant être émis vers une ou plusieurs destinations (console, fichier, eventlog, mail, SGBD...).

En cas d'erreur lors de l'exécution d'un script PowerShell, cet historique pourra nous aider à retrouver l'origine d'un problème plus rapidement.

Je vous propose d'aborder dans ce tutoriel les manipulations de base d'une gestion de log centralisé à l'aide de Log4NET version 1.2.10.

Spécial dédicace à chm69 ;-)

Site de l'auteur : http://laurent-dardenne.developpez.com/

Chapitres

1 L'existant

PowerShell fournit en standard les cmdlets d'écriture suivant :

```
gcm -verb write |?{$_.pssnapin.name -match "Microsoft.PowerShell"}
```

Write-Debug	Écrit un message de débogage vers la console, cette écriture dépend de la valeur de la variable *$DebugPreference*.
Write-Verbose	Écrit un message vers la console, cette écriture dépend de la valeur de la variable *$VerbosePreference*.
Write-Warning	Écrit un message d'avertissement, cette écriture dépend de la valeur de la variable *$WarningPreference*.
Write-Host	Écris un message dans la fenêtre du host.
Write-Output	Écris un message dans le pipeline courant.
Write-Error	Écris un message dans le pipeline d'erreur.

Le cmdlet **Set-Content** quant à lui écrit une chaîne ou un tableau de chaînes dans un fichier texte ASCII ou Unicode.

Au travers des exemples suivants, on peut constater que la destination se fait sur la console de PowerShell.exe :

```
write-warning "Warning"|% {Write-host "Test";$_}
```

```
AVERTISSEMENT : Warning
write-debug "Warning"|% {Write-host "Test";$_}
```

```
DÉBOGUER : Warning
write-output "Warning"|% {Write-host "Test: $($_)"}
```

```
Test: Warning
write-error "Warning"|% {Write-host "Test: $($_)"}
```

```
write-error "Warning"|% {Write-host "Test: $($_)"} : Warning
```

Aucun des cmdlets *Write-**, à part le *Write –Output*, ne peut être utilisé avec le pipeline c'est-à-dire redirigé vers une autre destination que la console.

Chaque cmdlet propose le paramètre ***–Debug*** :

« Génère des informations sur l'opération, destinées aux programmeurs. Ce paramètre n'a d'effet que dans les applets de commande qui génèrent des données de débogage. »

Son fonctionnement est indépendant du contenu de la variable *$DebugPreference* et les messages sont affichés en jaune sur la console.

Certaines exécutions du cmdlet **Stop-Service** peuvent afficher des warnings ou des erreurs non-bloquantes. On peut en déduire que les cmdlets précités ne sont pas dédiés à une gestion de log, mais sont à considérer comme une gestion de message paramétrable à destination de l'utilisateur final, voir comme une aide à la mise au point de scripts comme l'est le cmdlet **Set-PSDebug** ou **Trace-Command.**

On peut se satisfaire de ces possibilités, mais très vite les scripts vont se retrouver parasités par une gestion des logs qui, si on adresse plusieurs destinations, peuvent devenir difficile à maintenir :

```
$Date + " : Dossier(s) " + $dos_source + " non trouvé(s). Copie
impossible."  | Out-File $log -Append -NoClobber
```

Dans le cas où on veut tracer l'exécution sur la console et dans un fichier on peut envisager d'ajouter un segment de pipeline :

```
"$Date : $source introuvable. Copie impossible."|% {Write-Debug $_} ;$_} |
Out-File $log -Append -NoClobber
```

En revanche si l'on souhaite selon le contexte écrire dans le fichier, mais pas sur la console ou l'inverse cela s'avérera difficile bien que la combinaison suivante soit possible :

```
"$Date : $source introuvable. Copie impossible."|% {Write-Debug $_ ;$_} |%
{Write-Warning $_;$_}| Out-File $log -Append –NoClobber
```

On peut aussi regrouper les segments de 'trace' dans un filtre :

```
Filter Trace-Script([String] $Log){
 Write-Debug $_
 Write-Warning $_
 $_| Out-File $log -Append –NoClobber
}

 #Variables globales de paramètrage
$DebugPreference="SilentlyContinue"
$WarningPreference="SilentlyContinue"

"$Date : $source introuvable. Copie impossible."| Trace-Script $log
```

Malheureusement mon approche très simpliste, je vous l'accorde, offre peu de souplesse et nécessitera de nombreux paramètres et scripts additionnels pour répondre aux différents besoins. C'est ici que Log4NET entre en scène puisqu'il offre une solution clé en main, est paramétrable et offrant de nombreuses fonctionnalités.

Si vous ne connaissez pas du tout Log4NET je vous recommande la lecture de cette excellente introduction (http://lutecefalco.developpez.com/tutoriels/dotnet/log4net/introduction/) qui vous permettra de comprendre au mieux la suite du présent tutoriel.

1.1 Le principe du framework

J'ai cité le cmdlet **Trace-Command** qui est plutôt dédié aux développeurs de cmdlet. Son rôle est de tracer l'exécution d'un Scriptblock. Il s'appuie en interne sur les classes Trace et TraceListener (http://msdn.microsoft.com/fr-fr/library/system.diagnostics.trace.aspx).

C'est le seul à ma connaissance qui utilise le système de trace de .NET, on paramètre les destinations et le niveau d'information à tracer à l'aide du cmdlet **Set-TraceSource.** Dans ce cas on tracera l'exécution de nombreuses instructions. On peut consulter la liste des traces disponible par **Get-TraceSource**.

Un exemple :

```
Get-TraceSource|Sort name
Set-Tracesource -name cmdlet*  -Option All  -PSHost
gcm -verb read |?{$_.pssnapin.name -match "Microsoft.PowerShell"}
 #... affichage des traces
Set-TraceSource -name cmdlet* -RemoveListener *
```

Ce cmdlet trace l'exécution du code interne à PowerShell. Log4NET peut servir à mettre en place un mécanisme identique, d'un niveau de détail moindre, cette fois-ci sur le déroulement des scripts utilisateurs. Voyons le fonctionnement de ses méthodes de base.

2 Initialisation du Framework

Le moteur de Log4NET est contenu dans une seule DLL, ce qui facilitera le déploiement :

```
cd VotreChemin
$Log4NetHome=$PWD
# charge la librairie log4net
[void][Reflection.Assembly]::LoadFile( "$Log4NetHome\log4net.dll")
```

Par défaut le projet C# génère cet assembly avec un nom fort, il est donc possible de l'installer dans le GAC. Vérifiez s'il n'y est pas déjà installé et si le numéro de version est identique.

Le système de log est configuré par défaut au chargement de cet assembly.

Ensuite on a besoin d'un objet logger qui est l'objet de base du système de log, c'est à l'aide d'un logger qu'on écrit un message. Affichons la liste des loggers :

```
[log4net.LogManager]::GetCurrentLoggers()
```

```
#RAS
```

Les loggers sont organisés en interne dans une hiérarchie, ce qui permettra de placer un logger par script puis d'indiquer quel niveau d'imbrication on souhaite, nous verrons plus tard cette possibilité.

Pour le moment, essayons de récupérer le logger créé par défaut, l'instruction précédente ne nous ayant rien renvoyé, vérifions s'il existe un repository :

```
[log4net.LogManager]::GetAllRepositories()
```

```
EmittedNoAppenderWarning  : False
Root                      : log4net.Repository.Hierarchy.RootLogger
LoggerFactory             : log4net.Repository.Hierarchy.DefaultLoggerFactory
Name                      : log4net-default-repository
Threshold                 : ALL
RendererMap               : log4net.ObjectRenderer.RendererMap
PluginMap                 : log4net.Plugin.PluginMap
LevelMap                  : log4net.Core.LevelMap
Configured                : False
Properties                : {}
```

Un repository ou référentiel sert à organiser les loggers. Par défaut il existe un repository par processus, c'est-à-dire par domaine d'application. Nous nous contenterons de cette configuration par défaut.

L’appel à *ResetConfiguration* à bien crée quelque chose, mais pas notre logger. On doit le créer de la manière suivante :

```
$Log = [log4net.LogManager]::GetLogger("LogPowerShell")
```

La méthode *GetLogger* récupère ou crée un logger s'il n'en existe pas déjà un de même nom.

Tentons d’écrire un message, en fait on émet un événement qui est pris en charge par le système de log :

```
$Log.Info("Premier Message.")
#ras
```

L’instruction précédente n’affiche rien, car bien que l’on ait un logger nous n’avons pas d’appender associé. Un appender est une destination c'est-à-dire un dispositif sur lequel on écrit/envoie un événement. Vérifions l’objet crée :

```
$Log
```

```
IsDebugEnabled  : False
IsInfoEnabled   : False
IsWarnEnabled   : False
IsErrorEnabled  : False
IsFatalEnabled  : False
Logger          : log4net.Repository.Hierarchy.DefaultLoggerFactory+LoggerImpl
```

La propriété *IsInfoEnabled*, en lecture seule, nous indique que la prise en compte des événements de type INFO n’est pas active.

Pour visualiser l’appender par défaut on le récupérer du repository par défaut :

```
$Repositories=[log4net.LogManager]::GetAllRepositories()
$DefaultRepository=$Repositories[0]
$DefaultRepository.Root.Appenders
#ras
```

Pour afficher notre message, on doit également configurer le logger par défaut :

```
[log4net.Config.BasicConfigurator]::Configure()
 #on vérifie de nouveau le contenu de l’appender par défaut
$DefaultRepository.Root.Appenders[0]
```

```
Target      : Console.Out
Threshold   :
ErrorHandler : log4net.Util.OnlyOnceErrorHandler …
```

La configuration par défaut crée une instance de **ConsoleAppender** qui écrit sur 'Console.Out'.

```
$R.Root.Appenders[0].Layout
```

```
ConversionPattern   :  %timestamp [%thread] %level %logger %ndc - %message%newline
…
```

Les messages de log sont formatés à l'aide d'un objet **PatternLayout** ayant le style *DetailConversionPattern.* Un objet **Layout** contient des indications de mise en page/formatage d'une chaîne de caractères.

Le système étant désormais configuré, l’affichage de notre message est bien pris en compte :

```
$Log.Info("Premier Message.")
```

```
1763015 [Pipeline Execution Thread] INFO LogPowerShell (null) - Premier Message.
```

Comme on le voit, on peut compléter notre message avec des champs prédéfinis. En revanche, notre logger ne référence aucun appender encore moins celui créé par défaut bien qu'il l'utilise :

```
$Log.logger
```

```
Parent          : log4net.Repository.Hierarchy.RootLogger
Additivity      : True
EffectiveLevel  : DEBUG
Hierarchy       : log4net.Repository.Hierarchy.Hierarchy
Level           :
Appenders       : {}
Name            : LogPowerShell
Repository      : log4net.Repository.Hierarchy.Hierarchy
```

Avec la configuration par défaut, et puisque les loggers sont hiérarchisés, notre logger *$Log* utilise son logger parent :

```
$Log.Logger.parent.Name
```

```
root
```

```
$Log.Logger.parent.Hierarchy.name
```

```
log4net-default-repository
```

```
$log.Logger.parent.Appenders
```

```
Target       : Console.Out
Threshold    :
ErrorHandler : log4net.Util.OnlyOnceErrorHandler
…
```

Pour réinitialiser le système de logging avec la configuration par défaut faire :

```
[log4net.LogManager]::ResetConfiguration()
[log4net.Config.BasicConfigurator]::Configure()
```

2.1 Paramétrage de l'initialisation avec un FileAppender

La documentation du code source va nous être ici très utile, consulter de préférence le fichier .CHM fournis, car il intègre un menu *Recherche.*

Configurons le système de log en précisant cette fois-ci en destination un fichier de log.

La classe **FileAppender** gère les fichiers de log.

```
 [log4net.LogManager]::ResetConfiguration()

 #crée un appender fichier
$File = new-object log4net.Appender.FileAppender
$File.Name="FichierDeLog"
$File.File="$Pwd\PS-Log1.txt"
 #on ajoute dans le fichier s'il existe
$File.AppendToFile=$True

$logpattern = "%date [%thread] %-5level [%x] - %message%newline"
$File.Layout=new-object log4net.Layout.PatternLayout($logpattern)
```

Une fois l'appender crée et configuré, on configure le système de log de la manière suivante :

```
[log4net.Config.BasicConfigurator]::Configure($File)
  #la méthode GetLogger se comporte comme un singleton
$Log = [log4net.LogManager]::GetLogger("LogPowerShell")
$Log.Info("Premier Message.")
```

Mais la dernière ligne provoque une exception, car dès qu'on modifie les propriétés de configuration d'un ***Appender***, on doit impérativement appeler la méthode *ActivateOptions* sinon l'objet est dans un état indéterminé et ne peut être utilisé :

```
…
$File.ActivateOptions()
[log4net.Config.BasicConfigurator]::Configure($File)
…
```

Vous devez donc appeler cette méthode après chaque modification. Ensuite pour visualiser les propriétés de notre appender par défaut :

```
$Root=([log4net.LogManager]::GetAllRepositories())[0].Root
$Root.GetAppender("FichierDeLog")
 #ou
[log4net.LogManager]::GetLogger("Root").Logger.Repository.GetAppenders()
```

Notez que la collection *Appenders* de notre logger est toujours vide :

```
$Log.logger.Appenders.Count
0
```

Vous noterez également que sans notre intervention de notre part, notre logger utilise la nouvelle configuration. Il utilise donc toujours le paramétrage par défaut, mais ici c'est le nôtre.

2.1.1 Modèle de verrouillage du fichier

En cas d'exécution multiple d'un script configurant un **FileAppender**, les initialisations provoqueront l'erreur suivante :

```
log4net : ERROR [FileAppender] Unable to acquire lock on file C:\Temp\PS-Log1.txt. Le processus ne peut pas accéder au fichier ' C:\Temp\PS-Log1.txt', car il est en cours d'utilisation par un autre processus.
```

Par défaut le modèle de verrouillage est exclusif et utilise la classe imbriquée **FileAppender.ExclusiveLock**, si vous comptez logger les événements de plusieurs scripts dans un même fichier de destination utilisez la classe imbriquée **FileAppender.MinimalLock**, autorisant l'écriture pour d'autres processus.

```
[log4net.LogManager]::ResetConfiguration()

$File = new-object log4net.Appender.FileAppender
$File.Name="FichierDeLog"
$File.File="$Pwd\PS-Log1.txt"
$File.AppendToFile=$True
 #crée une instance d'une classe imbriquée
$File.LockingModel=New-object log4net.Appender.FileAppender+MinimalLock
```

```
$logpattern = "%date [%thread] %-5level [%x] - %message%newline"
$File.Layout=new-object log4net.Layout.PatternLayout($logpattern)
$File.ActivateOptions()
[log4net.Config.BasicConfigurator]::Configure($File)
$Log = [log4net.LogManager]::GetLogger("LogPowerShell")
$Log.Info("Premier Message.")
```

2.2 Création d'un ConsoleAppender

Comme indiqué au début de ce tutoriel, l'intérêt est de pouvoir tracer vers plusieurs destinations, par exemple dans un fichier et sur la console. Ajoutons à notre logger un **ConsoleAppender** :

```
$ConsoleAppender = new-object log4net.Appender.ConsoleAppender
$ConsoleAppender.Name="LogConsole"
$logpattern2 = "%date{dd-MM-yyyy } %-5level [%x] - %message%newline"
$ConsoleAppender.Layout=new-object `
 log4net.Layout.PatternLayout($logpattern2)
$ConsoleAppender.ActivateOptions()
$Log.Logger.AddAppender($ConsoleAppender)
$Log.Info("Second Message.")
```

```
2009-01-06 19:21:21,305 INFO  [(null)] - Second Message.
```

```
$F=([log4net.LogManager]::GetAllRepositories())[0].Root.Appenders[0].File
Type $F
```

```
2009-01-06 19:20:21,540  INFO  [(null)] - Premier Message.
2009-01-06 19:21:21,305  INFO  [(null)] - Second Message.
```

Revenons à notre collection *Appenders* :

```
$Log.logger
```

```
…
Appenders     : {LogConsole}
Name          : LogPowerShell
…
```

```
$Log.Logger.GetAppender("LogConsole")
```

Notre appender y a bien été ajouté, si l'on vide cette collection :

```
$Log.logger.RemoveAllAppenders()
$Log.Info("Troisième Message.")
```

L'appender de la configuration par défaut restera actif :

```
Type $F
```

```
2009-01-06 19:20:21,540  INFO   [(null)]  -  Premier Message.
2009-01-06 19:21:21,305  INFO   [(null)]  -  Second Message.
2009-01-06 19:22:21,633  INFO   [(null)]  -  Troisième Message.
```

2.3 Paramétrer le niveau de détail des logs

Chaque événement (LoggingEvent) est associé à un niveau de log. Par exemple la méthode $Log.Info soumet au système de log un événement de niveau INFO.

Règle : **Une requête de log de niveau *L* dans un logger ayant le niveau *K* que ce niveau soit assigné ou hérité, est activé si *L* est supérieure ou égal à *K*.**

Les niveaux de log sont donc ordonnés comme ceci :

```
OFF < DEBUG < INFO < WARN < ERROR < FATAL < ALL
```

Visualisons ce comportement au sein ‘une nouvelle session PowerShell :

```
cd votreChemin
$Log4NetHome=$PWD
[void][Reflection.Assembly]::LoadFile( "$Log4NetHome\log4net.dll")
[log4net.LogManager]::ResetConfiguration()

$Log = [log4net.LogManager]::GetLogger("LogPowerShell")
[log4net.Config.BasicConfigurator]::Configure()

$logpattern = "%-5level - %message%newline"
$Layout=new-object log4net.Layout.PatternLayout($logpattern)
$Root=([log4net.LogManager]::GetAllRepositories())[0].Root.Appenders[0]
$Root.Layout=$Layout
$Root.ActivateOptions()

function TestLevel{
 Write-host "Level $($Log.Logger.Level)" -for green
 $Log.Debug("Level Debug.")
 $Log.Info("Level Info.")
 $Log.Warn("Level Warn.")
 $Log.Error("Level Error.")
 $Log.Fatal("Level Fatal.")
}

 #modifie le niveau d'affichage
$Log.Logger.Level=[log4net.Core.Level]::Warn
 #Les messages Debug et Info ne sont plus affichés
TestLevel
 #seuls les messages Fatal seront affichés
$Log.Logger.Level=[log4net.Core.Level]::Fatal
TestLevel
```

```
 #seuls les messages Error seront affichés
$Log.Logger.Level=[log4net.Core.Level]::Error
TestLevel
 #Plus aucun message n'est affiché
$Log.Logger.Level=[log4net.Core.Level]::Off
TestLevel
 #tous les messages sont affichés
$Log.Logger.Level=[log4net.Core.Level]::All
TestLevel
```

Comme on peut le voir, une requête *Info* vers un logger configuré avec le niveau ***Warn*** ne sera pas traitée par le système de log.

On peut également placer un seuil de log sur l'appender :

```
 #Repository par défaut
$Repository=([log4net.LogManager]::GetAllRepositories())[0]
 #Mapping d'un objet Level de type Info
$Repository.LevelMap["INFO"]
 #Appender par défaut
$Root=([log4net.LogManager]::GetAllRepositories())[0].Root.Appenders[0]
$Root.Threshold=$Repository.LevelMap["INFO"]
Testlevel
```

Configuré de cette façon, le logger au niveau *All* et l'appender au niveau *Info*, l'appel à la fonction *TestLevel* n'affichera pas les événements de log de niveau inférieur, ici ceux de niveau *Debug*. Dans ce cas c'est le niveau de l'appender qui primera, on filtre ainsi les événements.

Un logger possède deux membres nous informant du niveau de log :

```
$log.Logger.Level
$log.Logger.EffectiveLevel
```

Le membre *EffectiveLevel* renvoie le niveau réel de log en tenant compte de la hiérarchie des loggers, notion qu'on abordera prochainement.

2.4 Ajouter un niveau de détail personnalisé

Le mapping entre un nom de niveau et l'objet **Level** correspondant se situe dans un repository :

```
$Repository.LevelMap.AllLevels|Sort value -des
 #ou  $Log.logger.Hierarchy.LevelMap.AllLevels
```

```
Name                Value           DisplayName
----------          --------------  -----------
OFF                 2147483647      OFF
EMERGENCY            120000         EMERGENCY
…
DEBUG                 30000         DEBUG
FINE                  30000         FINE
```

```
FINER                 20000     FINER
TRACE                 20000     TRACE
…
ALL                2147483648   ALL
```

On s'aperçoit qu'il existe d'autres niveaux et certains sont de même valeur.

Ajoutons un niveau personnalisé :

```
$ScriptLevel= new-object log4net.Core.Level 41000,"SCRIPT"
```

Ajoutons notre niveau dans le repository :

```
$Repository.LevelMap.Add($ScriptLevel)
$Repository.LevelMap.AllLevels|Sort value –des
```

On peut le déclencher ainsi :

```
$LogMain.Logger.Log($ScriptLevel, "test",$null);
```

Ou à l'aide d'une structure de type **LoggingEventData :**

```
$Data=New-object log4net.Core.LoggingEventData
$Data.Level=$Scriptlevel
$Data.Message="Usage de LoggingEventData"
$Data.TimeStamp=[System.DateTime]::Now
#$Data…
$LogMain.Logger.Log($Data);
```

Consultez l'aide en ligne pour le détail des champs de cette structure :

```
$pageEventData="/html/7c6788fe-efae-d45a-481e-ebda3fa8d0c4.htm"
```

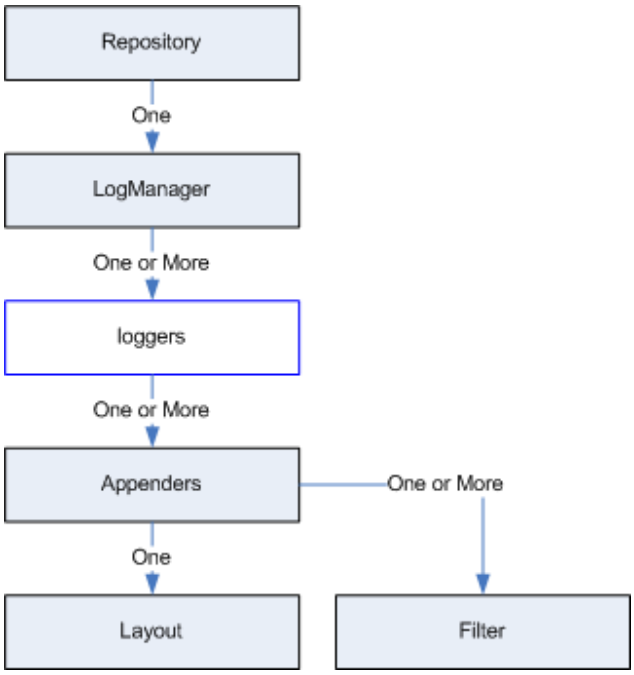
2.5 Description et usage des niveaux de log

D'après un document de Rob Prouse (http://www.alteridem.net/category/net/log4net/) :

FATAL	Pour les erreurs sérieuses qui peuvent provoquer l'arrêt du programme.
ERROR	Pour les erreurs qui peuvent corrompre des données ou provoquer un comportement erratique du programme.
WARN	Pour les erreurs prises en compte, mais pour lesquelles les développeurs veulent en garder une trace.
INFO	Log non verbeux du démarrage de sous-systèmes, synchronisations, appels distants (remoting), etc. Ne pas utiliser pour des logs verbeux ou dans des boucles pouvant ralentir le programme.
DEBUG	Pour des logs verbeux de mises au point de bas niveau, utilisé principalement pour aider les

	développeurs à détecter les erreurs. Peut-être employé dans les boucles, pour détailler le flux d'exécution du programme, etc.

Voici la hiérarchie des objets utilisés :



Extrais de "log4net XmlConfigurator Simplified" By Joseph Guadagno

(http://www.codeproject.com/KB/cs/log4net_XmlConfigurator.aspx)

Voir aussi : http://www.codeproject.com/KB/architecture/PipeDesign.aspx

3 Initialisation par fichier de configuration

Comme il est dit dans le SDK .NET à propos d'un fichier de configuration lié à une application,

« Les fichiers de configuration sont des fichiers XML qui peuvent être modifiés selon les besoins. Les développeurs peuvent utiliser les fichiers de configuration pour modifier des paramètres sans recompiler leurs applications. Les administrateurs peuvent utiliser les fichiers de configuration pour définir des stratégies qui affectent la façon dont les applications s'exécutent sur leurs ordinateurs.»

Sous PowerShell on ne peut pas associer ce type de fichier à un script, heureusement les concepteurs de Log4NET ont bien fait les choses, car il autorise à charger un fichier de configuration sans passer par le mécanisme d'association du Framewok .NET

Le chargement et l'initialisation à partir d'un fichier XML, se font comme ceci :

```
[log4net.LogManager]::ResetConfiguration()
```

```
$FileInfo=new-object System.IO.FileInfo "$PWD\PSLog.App.Config"
[log4net.Config.XmlConfigurator]::Configure($FileInfo)
```

Avec le fichier de configuration "PSLog.App.Config" contenant :

```
<?xml version="1.0" encoding="utf-8" ?>
<log4net>
  <appender name="AppenderPSLogConsole" type="log4net.Appender.ConsoleAppender">
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%-4timestamp [%thread] %-5level %logger %ndc - %message%newline" />
    </layout>
  </appender>
  <root>
    <level value="DEBUG" />
    <appender-ref ref="AppenderPSLogConsole" />
  </root>
</log4net>
```

Ensuite reste à récupérer dans le code, l'appender ainsi configuré :

```
$Log=[log4net.LogManager]::GetLogger("AppenderPSLogConsole")
```

L'initialisation par un fichier de configuration est plus souple, évite de nombreuses lignes de code et ne nécessite pas ou peu de maintenance du script en cas de modification de la configuration.

Le Framework log4net propose également la méthode statique *ConfigureAndWatch* qui charge une configuration et surveille la modification du fichier XML utilisé. Si cette surveillance, basée sur une instance **FileSystemWatcher**, détecte une modification alors la configuration est rechargée.

Voir aussi :

Les fichiers de configuration

http://msdn.microsoft.com/fr-fr/library/1xtk877y(VS.80).aspx

Travailler avec les fichiers de configuration en C#

http://nico-pyright.developpez.com/tutoriel/vc2005/configurationsectioncsharp/

3.1 Manipulation d'un fichier XML

PowerShell manipule un fichier XML de manière particulière en construisant un objet dont le contenu est une hiérarchie représentant celui du fichier XML. On retrouvera les balises sous forme de propriétés pouvant contenir un objet imbriqué ou une valeur.

3.1.1 Charger le fichier

Le raccourci ***[XML]*** associé au cmdlet **Get-Content** suffit :

```
[xml] $Config=get-content "$PWD\PSLog.App.Config"
```

3.1.2 Se placer sur une balise

La première balise se nomme log4net :

```
<log4net>
```

```
  <appender name="AppenderPSLogConsole" type="log4net.Appender.ConsoleAppender">
    <layout type="log4net.Layout.PatternLayout">
```

Voyons le contenu de l'objet créé

```
$config|Get-Member -MemberType Property
```

```
   TypeName: System.Xml.XmlDocument

Name     MemberType   Definition
----     ----------   ----------
log4net  Property     System.Xml.XmlElement log4net {get;}
xml      Property     System.String xml {get;set;}
```

Utilisons la propriété de même nom que notre balise :

```
$config.log4net
```

```
appender              root
--------              ----
appender              root
```

On retrouve bien les balises imbriquées. Pour naviguer dans cette hiérarchie on réitère l'opération :

```
$config.log4net.appender|gm -membertype property
```

```
   TypeName: System.Xml.XmlElement

Name   MemberType   Definition
----   ----------   ----------
layout   Property   System.Xml.XmlElement layout {get;}
name     Property   System.String name {get;set;}
type     Property   System.String type {get;set;}
```

```
$config.log4net.appender
```

```
name                  type                                  layout
----                  ----                                  ------
AppenderPSLogConsole  log4net.Appender.ConsoleAppender      layout
```

Cette fois ci les propriétés *name* et *type* sont de type string et référencent les attributs de la balise <appender …> :

```
  <appender name="AppenderPSLogConsole" type="log4net.Appender.ConsoleAppender">
$config.log4net.appender.layout

$config.log4net.appender.layout.conversionpattern

…
```

3.1.3 Accès au contenu d'une balise

Les balises et attributs étant transformés en propriétés leur manipulation reste identique à celles des propriétés d'un quelconque objet :

```
$config.log4net.appender.layout.conversionpattern.value
```

```
%-4timestamp [%thread] %-5level %logger %ndc - %message%newline
```

```
$config.log4net.appender.layout.conversionpattern.value="Mon formatage"

$config.log4net.appender.layout.conversionpattern
```

```
value
-----
Mon formatage
```

3.1.4 Ajouter un nœud

Ajoutons au niveau supérieur les informations suivantes :

```
<logger name="Task">
   <level value="WARN"/>
 </logger>
```

Crée le nœud de type 'element' et ayant comme nom de balise "logger"

```
[XML.XmlNode]$Noeud=$Config.CreateNode("element","logger", "")
 #Récupére la racine
[XML.XmlElement] $Root = $Config.psbase.DocumentElement
 #ajoute le noeud 'sous' la racine
[void]$Root.AppendChild($Noeud)
 #Visualisation du nouveau noeud
$Config.Log4net

 #Crée un élément ayant comme nom de balise "level"
$ElementLEVEL = $Config.CreateElement("level")
[void]$Noeud.AppendChild($ElementLEVEL)
 #affiche le document XML
$Config.psbase.OuterXml
```

La liste des types de nœud http://msdn.microsoft.com/fr-fr/library/system.xml.xmlnodetype.aspx

Pour ajouter un commentaire :

```
[XML.XmlNode]$Commentaire=$Config.CreateNode("comment","", "")
$Config.log4net.logger.AppendChild($Commentaire)
$Config.Log4net.logger."#comment"="ceci est un commentaire"
```

3.1.5 Supprimer un nœud

On sélectionne d'abord le nœud à supprimer :

```
$logger=$Config.selectSingleNode('log4net/logger')
$logger -eq $null
```

```
False
```

Attention cette sélection est sensible à la casse :

```
$logger=$Config.selectSingleNode('log4net/LOGGER')
$logger -eq $null
```

```
True
```

Puis on appel la méthode *RemoveChild* en lui passant en paramètre le nœud à supprimer :

```
$Config.log4net.RemoveChild($logger)
$Config.log4net
```

3.1.6 Ajouter un attribut

On utilisera la méthode *SetAttribut* sur le nœud ou l'élément cible :

```
$Noeud.SetAttribute("name","Task")
$ElementLEVEL.SetAttribute("value","WARN")
```

3.1.7 Enregister un objet XML dans un fichier

On utilisera la méthode *Save* :

```
 #sauve le fichier sur disque
$Config.save("$PWD\PSLog.App.Config2")
```

Pour visualiser la structure on passe en paramètre le flux de la console courante :

```
$Config.save([console]::out)
```

Cet affichage formaté est plus lisible que celui de la méthode *OuterXml* :

```
$Config.psbase.OuterXml
```

Le fichier de configuration présenté sur la page web suivante, propose de logger les événements de niveau INFO et DEBUG dans 2 fichiers distincts :

http://www.codeproject.com/KB/trace/Logger_using_log4net.aspx

4 Finalisation du Framework

Le moteur de log doit être arrêté proprement *via* l'appel à la méthode statique *LogManager.Shutdown :*

```
[log4net.LogManager]::Shutdown()
```

Certains appenders ont besoin d'opérations de finalisation comme la fermeture de fichier et la libération de ressources système. Elle appelle donc à son tour la méthode *Close* sur tous les appenders déclarés après avoir vidé les possibles buffers (*flush*), notez que la méthode *ResetConfiguration* appel en interne *Shutdown.*

La suppression d'un logger se fera en lui affectant la valeur *$Null.* Le finaliseur d'un appender appelle la méthode *Close*.

Un problème potentiel auquel vous devrez peut être faire face est qu'avec un finaliseur on ne sait pas quand le ramasse-miettes se déclenchera et libérera notre objet (finalisation non déterministe). Dans ce cas un verrou sur un fichier peut être encore actif alors que l'on souhaite de nouveau y logger, mais à partir d'un autre script que celui venant de se terminer. Une solution sera de forcer une collecte du garbage collector :

```
[GC]::Collect([GC]::MaxGeneration)
```

5 La classe Layout

Dans les premiers exemples nous avons utilisé la chaîne suivante pour définir le formatage des messages :

```
$logpattern = "%date [%thread] %-5level [%x] - %message%newline"
```

Où les mots débutant par le signe pourcentage correspondent à :

%date	Date du jour sur la machine courante.
[%thread]	Le nom ou le numéro du thread.
%-5level	Le niveau courant. Complète à droite avec des espaces si le nom de niveau fait moins de 5 caractères.
[%x]	Affiche le NDC (nested diagnostic context) associé au thread qui a produit l'événement de log.
%message	Le message à proprement parlé.
%newline	Un retour chariot.

Pour le détail des autres propriétés utilisables, consultez la documentation de la classe **PatternLayout**, le script suivant charge et affiche directement la page en question :

```
#Get-HelpChm.ps1
# voir aussi :
#  http://msdn.microsoft.com/en-us/library/ms670169(VS.85).aspx
#  http://www.microsoft.com/technet/scriptcenter/topics/winpsh/pschm.mspx
#  http://support.microsoft.com/kb/209843

$Chm=@{Path=$PWD;File="log4net-help.chm"}
$Target="{0}\{1}" -F $Chm.path,$Chm.File
$pagePatternLayout="/html/032ea939-ef22-f088-fde5-f4f417a59e6f.htm"
HH.exe "mk:@MSITStore:$($Target)::$pagePatternString"
```

Notez qu'il existe plusieurs classes de formatage.

```
$pageLayouts="/html/b8923ad0-f31c-7181-9d4c-1a66c0c19b55.htm"
```

5.1 Ajouter une propriété de formatage personnalisée

On peut vouloir ajouter dans une chaîne de formatage une information récurrente issue de l'exécution du script et pas du système de Log4NET, par exemple construire la chaîne suivante :

```
"%-4timestamp [%CurrentScript] %InvocationLine %RunspaceID %newline"
```

Déclarons une propriété :

```
[log4net.GlobalContext]::Properties.Item("Bonjour")="GlobalBonjour"
 #modifie le layout de l'appender
$Root=([log4net.LogManager]::GetAllRepositories())[0].Root.Appenders[0]

$Pattern="%date %property{Bonjour} %-5level [%x] - %message%newline"

$Root.Layout.ConversionPattern=$Pattern
$Root.Layout.ActivateOptions()
```

```
 #construit une chaîne formatée
$Log.InfoFormat("{0} -> {1}","Test",$Root.Target)
```

Notez que le nom de la propriété est sensible à la casse.

La classe **Log4net.ThreadContext** autorise à déclarer des propriétés propres à un thread, leur portée étant dans ce cas locale. Ceci pourrait être, sous réserve de vérification, utilisé avec les jobs de la version 2.

5.2 Tracer une exception

```
$Sb={
 Trap {$Log.Info("Except",$error[0].Exception)}
  #Ici seul le log est affiché dans la console
 #Trap {$Log.Info("Except",$error[0].Exception); Continue}

 $i=0
 1/$i
 }
&$Sb
```

L'objet $Error[0] ne peut être passé en paramètre, car son type n'est pas identique à *System.Exception*. Comme il contient des informations sur le script, le détail de l'exception pourrait être tracé en adaptant la fonction *Resolve-Error* suivante :

http://blogs.msdn.com/powershell/archive/2006/12/07/resolve-error.aspx

6 Les filtres

Nous avons vu précédemment que l'on pouvait placer un seuil sur un appender afin qu'il filtre les événements reçus. Il existe d'autres filtres basés sur la valeur un niveau, d'une étendue de niveau ou encore sur le contenu d'une chaîne ou d'une propriété.

Ces filtres sont stockés dans une liste chaînée, le premier est accessible via la propriété *Filterhead* d'un appender. Il est donc possible de combiner plusieurs filtres.

Comme dit dans le tutoriel d'introduction (http://lutecefalco.developpez.com/tutoriels/dotnet/log4net/introduction/#LII.5.2) :

« *Si le texte est trouvé, le filter va accepter le message et le traitement va s'arrêter, le message sera loggé. Si le texte n'est pas trouvé, l'événement sera passé au filter suivant. S'il n'y a pas de filter suivant, l'événement sera implicitement accepté et le message sera loggé. Étant donné que nous ne voulons pas logger les messages ne contenant pas la chaîne 'database', nous devons utiliser le filter log4net.Filter.DenyAllFilter qui va rejeter tous les événements qui l'atteindront. Ce filter est uniquement utile à la fin de la chaîne de filters.* »

On réutilise la fonction TestLevel du chapitre intitulé *Paramétrer le niveau de détail des logs* :

```
[log4net.LogManager]::ResetConfiguration()
$Log = [log4net.LogManager]::GetLogger("LogPowerShell")
[log4net.Config.BasicConfigurator]::Configure()
```

```
$Root=([log4net.LogManager]::GetAllRepositories())[0].Root
TestLevel
```

Ajoutons un filtre sur une étendue qui journalise uniquement les événements compris entre Warn et Critical :

```
$Filtre = new-object log4net.Filter.LevelRangeFilter
$Filtre.LevelMin = [log4net.Core.Level]::Warn
$Filtre.LevelMax = [log4net.Core.Level]::Critical
$Filtre.ActivateOptions()
$Root.Appenders[0].AddFilter($Filtre)
TestLevel
  #Raz des filtres
$Root.Appenders[0].ClearFilters()
```

Testons un filtre basé sur une expression régulière :

```
$Filtre2 = new-object log4net.Filter.StringMatchFilter
  #on recherche dans le texte des logs le mot Main
  #L'expression est sensible à la casse : "^(.*)[mM]ain(.*)$"
$Filtre2.RegexToMatch ="^(.*)Main(.*)$"
  #pour inverser la condition, faire : $Filtre2.AcceptOnMatch=$False
$Filtre2.ActivateOptions()
$Root.Appenders[0].AddFilter($Filtre2)
$Msgs=@("Test Main","Test Main.Script","Test Script","Test Demain")
$Msgs |ForEach {$Log.Warn($_) }
```

Comme dit précédemment on doit ajouter un filtre de terminaison afin de rejeter tous les événements qui passent au travers de la chaîne de filtrage. Sans ce filtre tous les événements sont loggés, même si les filtres précédents réussissent :

```
$Filtre3 = new-object log4net.Filter.DenyAllFilter
$Root.Appenders[0].AddFilter($Filtre3)
$Msgs |ForEach { $Log.Warn($_) }
```

7 Utiliser DebugView

Il est possible d'envoyer des événements de log vers un outil de visualisation comme l'application DebugView de SysInternal, laissant ainsi la possibilité de tracer dans une autre fenêtre que la console qui elle restera dédiée à l'affichage de messages destinés à l'utilisateur.

Télécharger DebugView : http://technet.microsoft.com/fr-fr/sysinternals/bb896647(en-us).aspx

```
  #         !!!!!!!!!! modifier le chemin d'accès !!!!!!!!!!
&"Dbgview.exe"
Start-Sleep 2
 #envoi un message
[int] $level=1
```

```
[string] $category="Warning"
[string] $message="Test"
[System.Diagnostics.Debugger]::Log($level, $category, $message)

 #Affiche le contenu de $A en chaîne de caractères uniquement
$A=Dir
[System.Diagnostics.Debugger]::Log(2,"Commentaire", $A)

  #On termine le process DebugView en lui envoyant le message Close
(get-process dbgview).CloseMainWindow()
  #dans ce cas l'exécution du code suivant ne pose pas de problème
[System.Diagnostics.Debugger]::Log(2,"Commentaire", $A)
```

A noter que les appels à *Debug.Write* au sein d'un exécutable sont également récupérés par DbgView.

Voici une fonction affichant le détail d'un objet :

```
function Write-ObjectProperties (
        $From, $PropertyName ="*",
        [Switch] $Pipeline, [Switch] $Debug)
{ #Affiche le contenu de toutes les propriétés d'un objet
  # $From         : l'objet à interroger,
  # $PropertyName : le nom de la propriété, ce nom peut contenir
  #                 des jokers,
  # $Pipeline     : indique si le résultat est émis dans le pipeline
  # $Debug        : indique si le résultat est envoyé vers le debugger
  #Exemples :
  # $a=dir
  # Write-Properties $a[-1]
  # Write-Properties $a[-1] -Pipeline |%{Write-host $_ -fore DarkGreen}
  # Res=Write-Properties $a[-1] -Debug -Pipeline

  foreach ($p in Get-Member -In $From -MemberType *Property `
                -Name $propertyName|Sort name)
  {
    $Result ="$($P.Name) : $($From.$($P.Name))"

    if ( $Pipeline.IsPresent)  {$result}
    else  {Write-Host $Result}
    if ( $Debug.IsPresent)
    {  # CommandLineParameters en PS v2 seulement
      #$Source=$MyInvocation.CommandLineParameters."From".ToString()
```

```
      [int] $level=1
      [string] $category="WP-$Source"
       #Inopérant si aucun débugger n'est actif
      [System.Diagnostics.Debugger]::Log($level, $category, $Result)
    }
  }
}
```

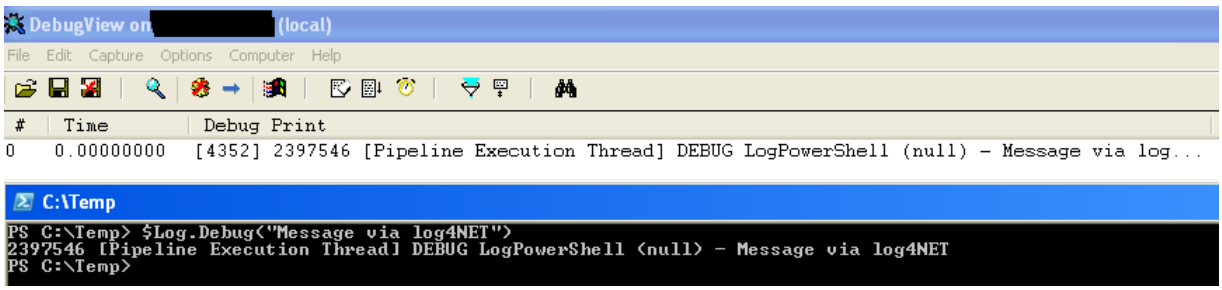
A noter que l'instruction *$MyInvocation.CommandLineParameters* n'est valide qu'à partir de la CTP2 de la version 2 de PowerShell.

7.1 L'appender OutputDebugStringAppender

On peut combiner l'usage de log4net et de DbgView de la manière suivante :

```
 # Si le logger autorise le niveau Debug on construit et
 # affiche le message suivant
if ($Log.IsDebugEnabled)
 {
   $DbgAppender = new-object log4net.Appender.OutputDebugStringAppender
   $logpatternDbg=
    "%-4timestamp [%thread] %-5level %logger %ndc - %message%newline"
   $DbgAppender.Layout=new-object `
      log4net.Layout.PatternLayout($logpatternDbg)
   $DbgAppender.Threshold = [log4net.Core.Level]::Debug
    #Récupére un objet logger et lui ajoute un appender dédié au debug
   $Log.Logger.AddAppender($DbgAppender)
    # **** modifier le chemin d'accès ****
   &"Dbgview.exe"
   $Log.Debug("Message via log4NET")
  }
```

Ce qui nous donnera, avec un ConsoleAppender par défaut, le résultat suivant :



La trace est bien émise vers les 2 appenders contenus dans le logger, le timestamp étant identique.

8 Hiérarchie nommée

Les instances des loggers suivent une certaine organisation hiérarchique. Elle est semblable à celle des namespaces dans .NET. Par exemple si deux loggers sont définis comme **a.b** et **a.b.c** alors le logger **a.b** est l'ancêtre du logger **a.b.c**.

Un logger est un ancêtre d'un autre logger si son nom suivi d'un point est un préfixe du nom du logger descendant (*Ancêtre.Descendant*). Un logger est un parent d'un logger enfant s'il n'y a aucun ancêtre entre lui-même et le logger descendant.

Par exemple, le code C# suivant nommera le logger en *System.IO.File* :

```
logger = LogManager.GetLogger(typeof(File));
```

Chaque logger hérite des propriétés de son logger parent. À l'origine de la hiérarchie se trouve le logger par défaut, qui s'appelle également le logger racine (root), tous les loggers en héritent. Il reste possible de nommer les loggers comme on le souhaite.

Créons une hiérarchie nommée de loggers :

```
cd VotreChemin
$Log4NetHome=$PWD
[void][Reflection.Assembly]::LoadFile( "$Log4NetHome\log4net.dll")
[log4net.LogManager]::ResetConfiguration()

 #Création d'une hiérarchie nommés de loggers
$LogMain = [log4net.LogManager]::GetLogger("Main")
$LogMainA = [log4net.LogManager]::GetLogger("Main.A")
$LogMainAB = [log4net.LogManager]::GetLogger("Main.A.B")
[log4net.Config.BasicConfigurator]::Configure()

function Log{
 $LogMain.Info("Message du logger : Main.")
 $LogMainA.Debug("Message du logger : Main.A.")
 $LogMainAB.Debug("Message du logger : Main.A.B")
}
```

Voyons le paramétrage des loggers créés :

```
Function ViewLoggers {
 #affiche quelques informations sur tous les loggers
 [log4net.LogManager]::GetCurrentLoggers()|`
  % {$_.Logger}|`
  Select
Name,Additivity,@{n="Parent";e={$_.Parent.Name}},Level,EffectiveLevel|`
  Format-List
}
ViewLoggers
```

```
Name          : Main.A
```

```
Additivity  : True
Parent      : Main

Name        : Main
Additivity  : True
Parent      : root

Name        : Main.A.B
Additivity  : True
Parent      : Main.A
```

Tous les loggers possèdent un parent et l'héritage est activé pour tous.

Le logger Main possède un appender console par défaut, les 2 autres loggers ne possèdent pas d'appender et hérite donc de celui de Main. Visualisons ce comportement :

```
Log
```

```
1328 [Pipeline Execution Thread] INFO Main (null) - Message du logger  : Main.
1375 [Pipeline Execution Thread] DEBUG Main.A (null) - Message du logger  : Main.A.
1375 [Pipeline Execution Thread] DEBUG Main.A.B (null) - Message du logger  : Main.A.B
```

Dans ce cas tous les loggers utilisent le seul appender déclaré dans le logger Main et toutes les demandes de création de log sont bien prises en compte. Maintenant, rompons le comportement d'addition :

```
$LogMainA.Logger.Additivity = $false
Log
```

```
1526437 [Pipeline Execution Thread] INFO Main (null) - Message du logger  : Main.
```

```
ViewLoggers
```

Comme vous le constatez, ici seul le logger Main gère les demandes de création de log, bien que la propriété *Additivity* du logger Main.A.B soit à *$True.*

```
$LogMainA.Logger.Additivity = $true
$LogMainAB.Logger.Additivity = $false
Log
```

```
2432812 [Pipeline Execution Thread] INFO Main (null) - Message du logger  : Main.
2432812 [Pipeline Execution Thread] DEBUG Main.A (null) - Message du logger  : Main.A.
```

8.1 Régle d'addition des appenders

La sortie d'une instruction de log d'un logger X est dirigée sur tous les appenders de X et ceux de ses ancêtres. C'est la signification du terme "additivité d'appender". Cependant, si un ancêtre d'un logger X, nommé Y, a la propriété d'additivité affectée à *$False*, Alors la sortie de X sera dirigée vers tous les appenders de X et ses ancêtres y compris Y mais pas vers les appenders ancêtres d'Y. Les loggers ont leur propriété d'additivité affectée à *$True* par défaut.

8.2 Création d'un ColoredConsoleAppender

Comme son nom le laisse supposer cet appender affiche sur la console des messages coloriés :

```
$ColoredConsoleAppender =
   new-object log4net.Appender.ColoredConsoleAppender
$ColoredConsoleAppender.Name="LogColoredConsole"
```

Vous devez créer une instance de la classe imbriquée **LevelColors** afin d'associer un niveau d'information à une couleur puis appeler la méthode *AddMapping* en lui passant en paramètre cette nouvelle instance :

```
$Colors = new-object log4net.Appender.ColoredConsoleAppender+LevelColors
#$Colors.BackColor= [log4net.Appender.ColoredConsoleAppender+Colors]::Blue
$Colors.ForeColor=[log4net.Appender.ColoredConsoleAppender+Colors]::Cyan
$Colors.Level=[log4net.Core.Level]::Info
$ColoredConsoleAppender.AddMapping($Colors)

#$Colors.BackColor= [log4net.Appender.ColoredConsoleAppender+Colors]::Blue
$Colors.ForeColor=[log4net.Appender.ColoredConsoleAppender+Colors]::Yellow
$Colors.Level=[log4net.Core.Level]::Debug
$ColoredConsoleAppender.AddMapping($Colors)
```

Ajoutons cet appender au logger Main.A :

```
$logpattern2 = "%date{dd-MM-yyyy } %-5level [%x] - %message%newline"
$ColoredConsoleAppender.Layout=
 new-object log4net.Layout.PatternLayout($logpattern2)
$ColoredConsoleAppender.ActivateOptions()
$LogMainA.Logger.AddAppender($ColoredConsoleAppender)
Log
```

Si vous n'avez pas modifié les propriétés *Additivity*, vous devriez avoir le résultat suivant :

```
2665937 [Pipeline Execution Thread] INFO Main (null) - Message du logger  : Main.
16-01-2009  DEBUG [(null)] - Message du logger  : Main.A.
2665937 [Pipeline Execution Thread] DEBUG Main.A (null) - Message du logger  : Main.A.
```

Désormais nous avons 2 appenders de déclarés, celui par défaut sur le root et celui sur Main.A.

Testons à nouveau la rupture de la hiérarchie des loggers :

```
[log4net.LogManager]::GetCurrentLoggers()|% {$_.Logger.Additivity=$True}
$LogMainA.Logger.Additivity = $false
ViewLoggers
Log
```

```
3145875 [Pipeline Execution Thread] INFO Main (null) - Message du logger  : Main.
16-01-2009  DEBUG [(null)] - Message du logger  : Main.A.
16-01-2009  DEBUG [(null)] - Message du logger  : Main.A.B
```

Le logger Main utilise toujours son appender, mais l'appender Main.A.B utilise celui de son ancêtre Main.A. Enfin désactivons le logger Main.A.B :

```
$LogMainA.Logger.Additivity = $true
$LogMainAB.Logger.Additivity = $false
ViewLoggers
Log
```

Ce comportement couplé par exemple avec un appender OutputDebugStringAppender, permettrait de tracer dans un debugger tout ou partie d'un traitement en ne modifiant que le

fichier de configuration. Sous réserve de copier *DebugView.exe* sur la machine cible sinon un **FileAppender** suffira.

On peut aussi imaginer de créer une hiérarchie de loggers dédiée aux logs fonctionnels et une autre dédiée aux logs de debug.

8.3 Hiérarchie du niveau de détail des loggers

La propriété *level* d'un logger peut ne pas être affectée, dans cas la règle suivante d'héritage s'applique :

Le niveau hérité (*inherited level)* d'un logger X est égal au premier niveau non nul dans la hiérarchie des loggers, commençant à X et remontant dans la hiérarchie vers le logger racine.

Un exemple :

```
 #on rétablie la hiérarchie
 #Additivity n'a pas d'influence sur l'héritage de niveau de log
[log4net.LogManager]::GetCurrentLoggers()|% {$_.Logger.Additivity=$True}
ViewLoggers
```

```
Name        : Main.A
Additivity  : True
Parent      : Main
Level       :
EffectiveLevel : DEBUG

Name        : Main
Additivity  : True
Parent      : root
Level       :
EffectiveLevel : DEBUG

Name        : Main.A.B
Additivity  : True
Parent      : Main.A
Level       :
EffectiveLevel : DEBUG
```

Notez que l'on peut placer un seuil sur une hiérarchie :

```
$LogMainAB.Logger.Hierarchy.Threshold=[log4net.Core.Level]::info
Log
```

L'affectation précédente modifie le seuil de toute la hiérarchie et pas seulement celui du logger LogMainAB. Le niveau affecté à cette propriété prime sur la propriété *level* d'un logger. Le seuil par défaut d'une hiérarchie est ALL.

```
$LogMainAB.Logger.Hierarchy.Threshold=[log4net.Core.Level]::All
```

Modifions le niveau du logger LogMainA

```
$LogMainA.logger.Level=[log4net.Core.Level]::Info
viewloggers
Log
```

```
670453 [Pipeline Execution Thread] INFO Main (null) - Message du logger  : Main.
```

Cette fois on intervient sur le niveau du logger et pas sur le niveau global de la hiérarchie ou sur le niveau particulier d'un appender.

Le logger LogMainAB héritera de son ancêtre puisque sa propriété *level* a la valeur *$null*.

Rétablissons la situation d'origine :

```
$LogMainA.logger.Level=$null
```

Si vous indiquez un seuil sur un appender d'un logger d'une hiérarchie, ce seuil primera sur tous les autres niveaux définis :

```
($LogMainA.logger.Appenders)[0].Threshold=[log4net.Core.Level]::Info
Log
2403750 [Pipeline Execution Thread] INFO Main (null) - Message du logger  : Main.
```

8.4 Création d'un EventLogAppender

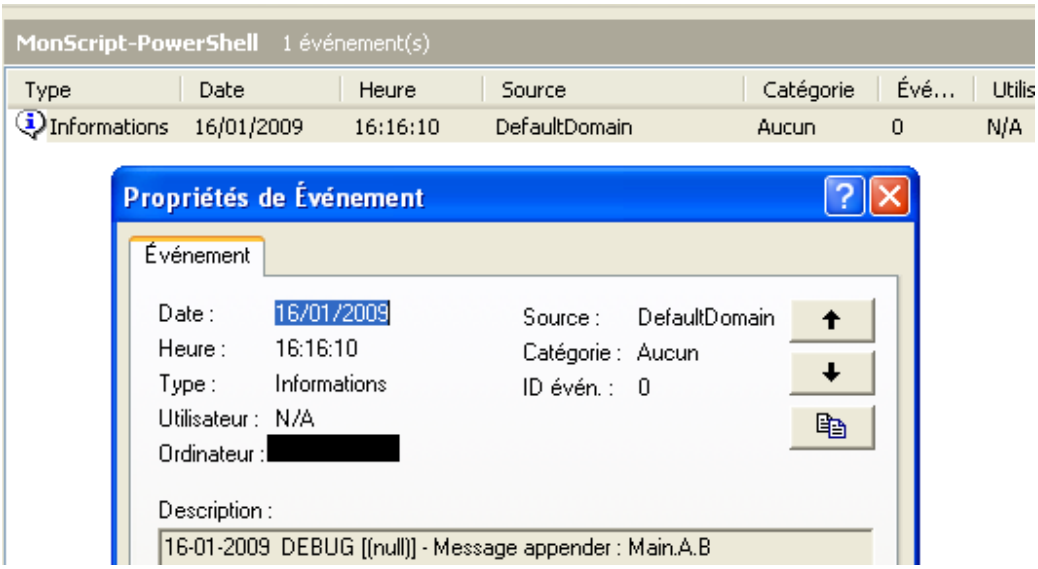
Le Framework Log4NET crée automatiquement, s'il n'existe pas déjà, le journal d'événement précisé dans la propriété *LogName.*

```
[log4net.LogManager]::GetCurrentLoggers()|% {$_.Logger.Additivity=$True}
$EventLogAppender = new-object log4net.Appender.EventLogAppender
$EventLogAppender.Name="LogEventLogAppender"

 #Nom du journal de log : Appplication,Systeme...
$EventLogAppender.LogName="MonScript-PowerShell"
$EventLogAppender.Layout=
 new-object log4net.Layout.PatternLayout($logpattern2)
$EventLogAppender.ActivateOptions()
$LogMainAB.Logger.AddAppender($EventLogAppender)
Log
```

Ainsi, chaque logger de la hiérarchie dispose de son appender pouvant ou non rediriger les logs vers différentes destinations.

Voici le contenu du journal d'événement nommé *MonScript-PowerShell* :



Il reste possible de définir le type d'événement d'une entrée du journal des événements, tout comme la gestion des couleurs d'une console colorisée, cette fois à l'aide de la méthode *AddMapping* utilisant la classe imbriquée ***EventLogAppender.Level2EventLogEntryType***.

Sachez aussi qu'un appender peut recevoir des événements de plusieurs logger.

8.5 Limite sous PowerShell

PowerShell ne propose pas la notion de namespace pour organiser du code, scripts et fonctions.

Les premiers sont persistants sur un disque et les seconds, une fois chargés, le sont dans le provider ***Function:***. Ce qui d'autre part rend difficile la construction d'un arbre des dépendances. On pourrait envisager de reprendre l'organisation des packages Java en se basant sur une arborescence disque et en remplaçant le caractère '\' par un point, mais ceci risque d'être à mon avis rigide, de plus PowerShell offre la possibilité d'exécution de code dynamique.

Pour le moment il sera difficile de bénéficier de tous les avantages de cette hiérarchisation des loggers.

9 Contexte

Le Framework Log4net prend en charge les applications multithreads, autorisant ainsi chaque thread à déclarer des informations de log propres à son contexte d'exécution : le nom du client, l'id du thread, etc. On peut donc placer des données contextuelles dans différentes portées.

Sous PowerShell V1 on ne rencontrera pas pour le moment ce cas de figure, à moins d'utiliser des Runspaces. En revanche on peut utiliser certaines classes manipulant la notion de contexte pour définir une information propre à un script ou une fonction.

Le principe est de définir globalement une macro (un champ prédéfini utilisé dans un objet layout) qui sera référencée dans une chaîne de formatage, puis de modifier provisoirement son contenu dans chaque script.

9.1 Les propriétés

Déclarons le pattern de message suivant :

```
$logpattern = "[%thread] %-5level [%x] %property{CallStack} - `
               %message%newline"
```

Où

- %thread représente le nom du thread qui à généré l'événement de log si le nom n'existe pas il contient l'id du thread,

- %x représente le NDC, nested diagnostic context, c'est à dire le contexte de thread courant, il est accessible par :

```
[log4net.ThreadContext]::Stacks["NDC"]
```

- **%property{CallStack}** une propriété personnelle que l'on défini soit globalement de la manière suivante :

```
[void][log4net.GlobalContext]::Properties["CallStack"]=
  "Propriété globale "
```

Soit localement, par exemple dans une fonction F :

```
[void][log4net.ThreadContext]::Properties["CallStack"]=
 "Propriété modifiée localement"
```

Si dans ce cas la fonction F appelle une autre fonction F2, qui ne modifie pas cette propriété, la fonction F2 utilisera la valeur renseignée par la dernière modification locale, c'est-à-dire ici celle ayant eu lieu dans la fonction F.

Une fois la fonction F terminée la valeur de cette propriété sera de nouveau celle définie dans le contexte global. Les propriétés du ThreadContext remplacent toutes propriétés du GlobalContext de même nom.

Avec ces contextes imbriqués vous pouvez ou non préciser certaines propriétés par exemple la pile d'appels :

```
$StrFrmt="Dans la fonction {0}`r`nPile = {1}"
Function Une {
 Write ($StrFrmt -F "Une",$(ParseStack $(Get-CallStack)))
 Deux }

Function Deux {
Write ($StrFrmt -F "Deux",$(ParseStack $(Get-CallStack)))
 Trois}

Function Trois {
 Write ($StrFrmt -F "Trois",$(ParseStack $(Get-CallStack)))
}
```

```
Une
```

```
Dans la fonction Une
Pile = Une
Dans la fonction Deux
Pile = Une:Deux
Dans la fonction Trois
Pile = Une:Deux:Trois
```

Les fonctions utilisées ici sont disponibles dans le fichier *PackageDebugTools.ps1*

Créons une propriété globale contenant la pile d'appel :

```
[void][log4net.GlobalContext]::Properties["CallStack"]=
 $(ParseStack $(Get-CallStack))
```

Dans ce cas son contenu est figé. On doit redéclarer son contenu dans chaque fonction de notre appel :

```
Function Une {
 [void][log4net.ThreadContext]::Properties["CallStack"]=`
 $(ParseStack $(Get-CallStack))
 $LogMain.Info("Dans la fonction Une")
 Deux }

Function Deux {
 [void][log4net.ThreadContext]::Properties["CallStack"]=`
  $(ParseStack $(Get-CallStack))
 $LogMain.Info("Dans la fonction Deux")
 Trois
 #Pour la ligne suivante, la pile d'appels affiche la ligne complète
 # $LogMain.Info("Dans la fonction Deux";Trois
}

Function Trois {
 [void][log4net.ThreadContext]::Properties["CallStack"]=`
  $(ParseStack $(Get-CallStack))
 $LogMain.Info("Dans la fonction Trois") }

$LogMain.Info("Appel la fonction Une ")
```

```
INFO  [CS->] - Appel la fonction Une
```

```
Une
```

```
INFO  [CS->Une] - Dans la fonction Une
INFO  [CS->Une:Deux] - Dans la fonction Deux
INFO  [CS->Une:Deux:Trois] - Dans la fonction Trois
```

```
$LogMain.Info("Fin")
```

```
INFO  [CS->] - Fin
```

Ce qui nous intéresse ici est d'obtenir dynamiquement la pile d'appels en cours. Pour cela on utilise une propriété dite active, *.i.e.* son contenu est mis à jour automatiquement lors de chaque appel.

9.2 Propriété active

Le code du Framework log4net fait appel en interne à la méthode *ToString* de l'objet contenu dans une propriété. On doit créer un objet personnalisé redéclarant sa méthode *ToString* :

```
$Obj = new-object System.Management.Automation.PSObject
$Obj|Add-Member -Force -MemberType ScriptMethod ToString {`
     $(ParseStack $(Get-CallStack)) }
[log4net.GlobalContext]::Properties["CallStack"]=$Obj

Function Une { $LogMain.Info("Dans la fonction Une")
 Deux }

Function Deux {  $LogMain.Info("Dans la fonction Deux")
 Trois
}

Function Trois {  $LogMain.Info("Dans la fonction Trois") }

$LogMain.Info("Appel la fonction Une ")
Une
$LogMain.Info("Fin")
```

Cela fonctionne puisque le membre personnalisé, la méthode *ToString*, a priorité sur celui d'origine et il n'est plus nécessaire de modifier en local la propriété "CallStack", elle est mise à jour automatiquement.

Pour cet exemple il vous faudra peut-être arrêter puis reconfigurer le Framework log4net.

9.3 Imbrication de contextes

Il existe une autre possibilité d'imbrication de contexte utilisant une pile LIFO basée sur la classe *System.Collections.Stack*.

Modifions le pattern d'affichage :

```
$logpattern2 =
 "[%thread] %-5level [CS->%property{CallStack}] (%x) -%message%newline"
$ColoredConsoleAppender.Layout.ConversionPattern=$logpattern2
$ColoredConsoleAppender.Layout.ActivateOptions()
```

Empilons pour le contexte courant, la ligne d'appel des fonctions :

```
Function Une {
```

```
 [void][log4net.ThreadContext]::Stacks["NDC"].Push(
        $($myinvocation.MyCommand))
$myinvocation.MyCommand
 $LogMain.Info("Dans la fonction Une")
 Deux
 [void] [log4net.ThreadContext]::Stacks["NDC"].Pop()
}

Function Deux {
 [void][log4net.ThreadContext]::Stacks["NDC"].Push(
        $($myinvocation.MyCommand))
$myinvocation.MyCommand
 $LogMain.Info("Dans la fonction Deux")
 Trois
[void][log4net.ThreadContext]::Stacks["NDC"].Pop()
}

Function Trois {
 [void][log4net.ThreadContext]::Stacks["NDC"].Push(
       $($myinvocation.MyCommand))
$myinvocation.MyCommand
 $LogMain.Info("Dans la fonction Trois")
[void][log4net.ThreadContext]::Stacks["NDC"].Pop()
}

$LogMain.Info("Appel la fonction Une ")
Une
$LogMain.Info("Fin")
```

On empile à l'aide de la méthode *Stacks["NDC"].Push* et on dépile avec *Stacks["NDC"]. Pop.*

Ici la propriété NDC référence une macro native du Framework, mais vous pouvez en créer d'autres :

```
"%-5level (%property{user}) -%message%newline"
[void][log4net.ThreadContext]::Stacks["user"].Push("Pierre")
```

Dans ces derniers exemples, bien que l'on utilise la classe **ThreadContext**, on se trouve toujours dans le même thread d'exécution de PowerShell, en tout cas dans le même Runspace !

```
$logpattern2 =
 "[%thread] %-5level RunspaceID:%property{RunspaceID} -%message%newline"
$ColoredConsoleAppender.Layout.ConversionPattern=$logpattern2
$ColoredConsoleAppender.Layout.ActivateOptions()
```

```
$Obj = new-object System.Management.Automation.PSObject
$Obj|Add-Member -Force -MemberType ScriptMethod ToString {`
     #Gets and sets the default runspace used to evaluate scripts.
 [Management.Automation.Runspaces.Runspace]::DefaultRunspace.InstanceID.`
ToString() }

[log4net.GlobalContext]::Properties["RunspaceID"]=$Obj
Une
 #la pile NDC est désormais inutilisée, mais la présence du code ne
 #provoque pas d'erreur.
#...
```

Voir aussi : http://logging.apache.org/log4net/release/manual/contexts.html

10 Les objets renderer

Les méthodes de log telles qu'*Info*, *Debug*, *Log*, etc peuvent recevoir en paramètre un objet imbriquant d'autres objets. Le rendu des objets conteneur des types suivant **Array**, **Hashtable** ou ceux implémentant les interfaces ***IEnumerable***, ***ICollection***, ***IEnumerator*** utiliserons un formatage particulier par défaut (consultez la documentation pour le détail). Les autres types d'objet utiliseront leur méthode *ToString*.

```
$File=Dir *.txt
$LogMain.Info($File)
```

```
…- Object[] {C:\Temp\Test.txt, C:\Temp\PS-Log1.txt}
```

```
$Ht=@{Nom="PowerShell";Version="1.0";Date=([System.DateTime]::Now)}
$LogMain.Info($Ht)
```

```
…- {Version=1.0, Date=27/01/2009 19:17:12, Nom=PowerShell}
```

Un objet renderer transforme un objet en une chaîne de caractères. Comme je l'ai précisé dans le chapitre intitulé *Tracer une exception*, le contenu d'un élément du tableau **$Error** ne pourra être traité.

Le Framework log4net utilise une interface pour accéder à la méthode *RendererObject* associée au type de l'objet que l'on souhaite tracer. Sous PowerShell il est possible d'ajouter à un objet une méthode, mais pas une interface. On doit donc coupler PowerShell et du code C# que soit par un compilateur ou par une génération dynamique de code à l'aide CodeDOM.

Créons une classe implémentant l'interface ***IObjectRenderer*** :

```
public class ObjectRenderer : IObjectRenderer
    {
        public void RenderObject(log4net.ObjectRenderer.RendererMap rendererMap,
object obj, System.IO.TextWriter writer)
        {
            if (rendererMap == null)
            { throw new ArgumentNullException("rendererMap");}
            if (obj == null)
            {
                writer.Write(SystemInfo.NullText);
                return;
            }
```

```
            ObjectRenderer Instance = obj as ObjectRenderer;
            if (Instance != null)
             //On laisse le scripteur redéclarer
             //la méthode ToString via add-member
            { writer.Write("*{0}*", Instance.ToString()); }
            else
            { writer.Write(SystemInfo.NullText); }
        }
    }
```

Ensuite, testons notre objet :

```
Cd votre chemin
$RndrPath=$Pwd
  # charge la librairie PSLog4NET (Renderer)
[void][Reflection.Assembly]::LoadFile("$RndrPath\PSLog4NET.dll")

$Renderer = new-object PSLog4NET.ObjectRenderer
$Repository = [log4net.LogManager]::GetRepository()
$Repository.RendererMap.Put([PSLog4NET.ObjectRenderer],`
                            (new-object PSLog4NET.ObjectRenderer) )
$Renderer|Add-Member -Force -MemberType ScriptMethod ToString {`
  Write-host "Write-host de Renderer"
   $(Rver)
}
$LogMain.Info($Renderer)
INFO  - *PSLog4NET.ObjectRenderer*
```

Notre méthode *RenderObject* est bien appelée par le Framework log4net, mais pas la méthode *ToString* redéclarée. Bien qu'elle le soit sous PowerShell :

```
$Renderer.ToString()
```

Dérivons maintenant notre classe de **PSObject** et plus de **System.Object**, le reste du code est identique à part le formatage :

```
public class PSObjectRenderer : PSObject,IObjectRenderer
    {
        public void RenderObject(log4net.ObjectRenderer.RendererMap rendererMap,
object obj, System.IO.TextWriter writer)
        {
           …
             { writer.Write("{0}", Instance.ToString()); }
           …
```

Testons notre nouvelle classe :

```
$PSRenderer = new-object PSLog4NET.PSObjectRenderer
$Repository.RendererMap.Put([PSLog4NET.PSObjectRenderer],`
                            (new-object PSLog4NET.PSObjectRenderer) )
$PSRenderer|Add-Member -Force -MemberType ScriptMethod ToString {`
   Write-host "Write-host de PSRenderer"
   $(Rver)
}
```

```
$LogMain.Info($PSRenderer)
```

```
Write-host de PSRenderer
INFO  - Microsoft.PowerShell.Commands.Internal.Format.FormatStartData
Microsoft.PowerShell.Commands.Internal.Format.GroupStartData
Microsoft.PowerShell.Commands.Internal.Format.FormatEntryData Microsoft.PowerShell.Comm
ands.Internal.Format.GroupEndData Microsoft.PowerShell.Commands.Internal.Format.FormatEndData
```

Cette nouvelle approche résout bien notre problème d'interface et de redéclaration de méthode, mais cette solution nous en apporte un nouveau. L'usage du cmdlet **Format-Table** dans la fonction *Resolve-Error* est inapproprié pour l'usage que l'on souhaite en faire. Ce cmdlet renvoie des directives de formatage destinées aux seuls cmdlet **Out-***.

Modifions notre méthode *ToString* afin de renvoyer une chaîne de caractères :

```
$PSRenderer|Add-Member -Force -MemberType ScriptMethod ToString {`
   Write-host "Write-host de PSRenderer"
   Rver|Out-String
}
$Error.clear()
$i=0
1/$i
```

```
Tentative de division par zéro.
...
```

```
$LogMain.Info($PSRenderer)
```

```
INFO  -

Exception               : System.Management.Automation.RuntimeException: Tentative de division …
TargetObject            :
CategoryInfo            : NonSpécifié : (:) [], RuntimeException
FullyQualifiedErrorId   : RuntimeException
ErrorDetails            :
InvocationInfo          : System.Management.Automation.InvocationInfo
…
```

Ainsi on obtient bien le résultat attendu. Pratique comme technique n'est-ce pas !

11 Tracer le Framework Log4Net

L'activation des traces du moteur de log4net s'appuie sur un fichier de configuration. Par défaut les traces s'afficheront au moins sur la console, qu'un fichier de configuration existe ou pas.

La recherche du fichier se fait dans le répertoire suivant :

```
[AppDomain]::CurrentDomain.BaseDirectory
```

C'est-à-dire $PSHome. On y créé le fichier *powershell.exe.config* contenant :

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
     <!-- Configure le fichier de trace des logs internes à Log4NET -->
    <trace autoflush="true">
      <listeners>
```

```
        <add name="textWriterTraceListener"
type="System.Diagnostics.TextWriterTraceListener"
initializeData="C:\Temp\log4net_InternalDebug.txt"/>
      </listeners>
    </trace>
   <!-- Trace le profil de PowerShell
      Attention les lignes suivantes sont commentées
    <sources>
      <source name="CommandDiscovery" switchValue="All" Options="All">
        <listeners>
          <add name="trace.log"
type="System.Diagnostics.TextWriterTraceListener"
initializeData="trace.log"/>
        </listeners>
      </source>
    </sources>
   -->
  </system.diagnostics>
  <appSettings>
   <!-- Autorise les logs internes à Log4NET -->
   <add key="log4net.Internal.Debug" value="true"/>
  </appSettings>
</configuration>
```

Pour arrêter les traces internes :

```
[log4net.Util.LogLog]::InternalDebugging=$False
  #Ou $True s'il n'existe pas de fichier de configuration
[log4net.Util.LogLog]::InternalDebugging=$True
```

12 Outils de visualisation des logs

L'application ASP .NET Log4NetDash est une solution intéressante et bon marché pour visualiser les logs générés, une licence développeur est proposée gratuitement.

Sur le site de l'éditeur, vous trouverez une démonstration du produit http://demo.l4ndash.com/ .

L'outil Baretail cité dans le tutoriel d'introduction de Log4NET permet de visualiser les logs dirigés vers un fichier au fur et à mesure de leur production, l'application surveillant les ajouts de nouvelles lignes.

Voir aussi

http://www.codeproject.com/KB/vb/LogViewer.aspx

13 Conclusions

N'hésitez pas à consulter la documentation de log4net, car je n'ai pas repris tous les détails des méthodes et propriétés utilisées dans ce tutoriel.

Je n'ai ni abordé de nombreuses classes pouvant enrichir ce système de log, car leur usage nécessite un langage compilé ni les autres appenders disponibles et encore moins les contextes multithread. Il vous faudra quelque temps pour étudier le paramétrage de ce Framework mais cela en vaut la peine.

Sachez que pour logger dans une base de données c'est à vous de créer la table, de plus la volumétrie de vos logs influencera le choix du SGBD le plus approprié et supporté par ADO .NET.

Enfin l'encapsulation de ce Framework reste à développer, sa manipulation s'en trouverait ainsi facilité. Bien que l'usage de fichiers de configuration soit suffisant pour de simples usages.

Le logging peut s'avérer appréciable pour les administrateurs en facilitant le diagnostic des problèmes dans les scripts en production. De plus, ces logs pourront être couplés avec des outils de supervision tout en facilitant la normalisation du format des messages.

Log4Net permet de concevoir un système de log centralisé, par exemple dans une base de données, ce qui sera bien utile avec l'arrivée des jobs proposés dans la version 2 de PowerShell.

14 Liens

La gestion correcte des logs en .Net : http://vincentlaine.developpez.com/tuto/dotnet/log/

Un autre système de log .NET basé sur un mécanisme bas niveau de Windows : Event Tracing for Windows (ETW) : http://www.codeplex.com/NTrace

Voir aussi sur le sujet ce tutoriel Delphi Win32 :

http://fsoriano.developpez.com/articles/etw/delphi/

Entrées de blog à propos des traces sous .NET, par Mike Rousos **(.NET BCL team)**

A Tracing Primer - Part I : http://blogs.msdn.com/bclteam/archive/2005/03/15/396431.aspx

A Tracing Primer - Part II (A) : http://blogs.msdn.com/bclteam/archive/2005/09/21/472015.aspx

A Tracing Primer - Part II (B) : http://blogs.msdn.com/bclteam/archive/2005/09/21/472021.aspx

A Tracing Primer - Part II (C) : http://blogs.msdn.com/bclteam/archive/2005/09/21/472049.aspx

Microsoft Enterprise Library Logging Block compared to Log4net :

http://weblogs.asp.net/lorenh/archive/2005/02/18/376191.aspx