Techniques avancées

Gestion des traces et des logs

Auteur :
Tarik Djebien

Mars 2012

Présentation

La conception de projets nécessite la mise en place de traces lors des développements, débogages mais aussi en phase de production. La plupart des projets utilisent des API de journalisation réalisées en interne et plus ou moins performantes. Les programmeurs placent des traces dans le code en utilisant la sortie standard et la classe statique System au moyen du code suivant :

```
System.out.println(''Une trace dans la console Java'') ;
```

Les outils de journalisation offrent de nombreux avantages aux programmeurs. Le service le plus utilisé est évidemment la mise au point du code lors des développement. Ils permettent également d'enregistrer les messages, d'envoyer des emails, de gérer des niveaux de traces ou autre, mais surtout par l'intermédiaire d'un fichier de configuration, de gérer à tout moment les traces. En effet, un outil de journalisation permet d'activer ou désactiver à tout moment certains messages en fonction de nos besoins, sans être obligé de reprendre tout le code.

L'API LOG4J

La bibliothèque Log4J est très répandue dans le monde Java et notamment Java EE. Cette API de journalisation permet de gérer les traces utilisateurs en combinaison avec l'API commons-logging de la fondation apache.
Nous allons commencer la mise en place de la journalisation pour notre projet en installant l'API LOG4J. La première étape consiste à télécharger et à installer les archives au format .jar dans notre répertoire de librairies /WEB-INF/lib les librairies suivantes :
- commons-logging.jar
- log4j-X.X.jar

Mise en place de l'API LOG4J

La librairie Log4J met à disposition du programmeur trois composants :
- Les Loggers qui permettent d'écrire les messages.
- Les Appenders qui permettent de sélectionner la destination des messages.
- Les Layouts qui permettent de mettre en forme les messages.

Logger

Le logger est l'entité de base qui est utilisée par la journalisation, il utilise la classe org.apache.log4J.logger. La déclaration d'un logger est réalisée dans chaque classe qui doit utiliser le système de journalisation. Nous pouvons par exemple reprendre la classe Exemple et ajouter la déclaration du Logger :

```
public class Exemple {
   //log4j
   private static final Logger logger = Logger.getLogger(''loggerName'') ;
}
```

L'obtention de l'instance du Logger est réalisée en appelant la méthode statique Logger getLogger(). Cette méthode prend en paramètre un nom de Logger de notre choix ou la référence directe à la classe.

```
Private static final Logger logger = Logger.getLogger(Exemple.class);
```

Ensuite, il est nécessaire de gérer le niveau de journalisation ou de priorité des messages. Ceci permet de représenter l'importance du message à journaliser. La classe org.apache.log4j.Level permet de gérer ces niveaux de messages. Un message n'est alors journalisé que si sa priorité est supérieure a celle du Logger effectuant la journalisation.

L'API Log4j définit cinq niveaux de logging classés par ordre d'importance :
- FATAL : ce niveau est utilisé pour une erreur grave pouvant provoquer l'arrêt de l'application.
- ERROR : ce niveau est utilisé pour une erreur qui empêche un fonctionnement important de l'application (requête SQL, copie de fichier...)
- WARN : ce niveau est utilisé pour un avertissement ou une trace.
- INFO : ce niveau est utilisé pour un message informatif.
- DEBUG : ce niveau très verbeux est utilisé pour des messages utilisés en phase de débogage.

La journalisation d'un message à un niveau donné se fait au moyen de la méthode log(priorité,message). Par exemple, nous pouvons déclencher une trace d'erreur des deux manières suivantes :

```
logger.error(''Erreur dans la classe Exemple.java fonction exemple '') ;
logger.log(Level.FATAL,''Erreur dans la classe Exemple fonction exemple '') ;
```

Appenders

Un Appender représente la cible d'un message, c'est à dire l'endroit où sera stocké ou affiché ce message. Les Appenders sont utilisés pour enregistrer les événements de journalisation. Ils sont représentés par l'interface org.apache.log4j.Appender et chaque Appender enregistre d'une manière spécifique les événements.
Il existe plusieurs types d'Appender afin de gérer les traces dans une base de données (org.apache.log4j.jdbc.JDBCAppender), par mail (org.apache.log4j.net.SMTPAppender), pour la console (org.apache.log4j.ConsoleAppender) ou encore dans un fichier (org.apache.log4j.FileAppender). Nous retrouvons donc des Apppenders pour la console, le système de fichiers, les sockets, le démon Unix syslog ou les composants graphiques. Nous pouvons reprendre notre classe Exemple.java afin de tracer le message d'erreur dans la console.

```
. . .

//Constructeur
public Exemple(){

   PatternLayout layout = new PatternLayout(''%d %-5p %c - %F:%L - %m%n'') ;
   ConsoleAppender stdout = new ConsoleAppender(layout) ;
   logger.addAppender(stdout) ;

   logger.error(''Erreur dans la classe Exemple.java constructeur '') ;
   logger.log(Level.FATAL ,'Erreur dans la classe Exemple.java constructeur'') ;

}
. . .
```

Le format défini avec le modèle (PatternLayout) permet de conserver l'heure et la date, le niveau d'erreur, le nom de fichier et le numéro de ligne de code correspondante au message lui même. Désormais si nous déclenchons un service qui utilise cette classe Exemple.java, les traces sont affichés dans la console Système :

```
2012-03-13 16:24:12,345 ERROR modele – Exemple.java:33 – Erreur dans la classe Exemple.java constructeur
2012-03-13 16:24:12,445 FATAL modele – Exemple.java:34 – Erreur dans la classe Exemple.java constructeur
```

La classe PatternLayout permet de gérer la mise en forme des messages en sortie. Cette classe permet d'informer les développeurs de données utiles mais demande parfois d'importantes ressources en fonction du détail des informations tracées.

Layouts

Les layouts sont utilisés pour la mise en forme des événements de journalisation. Ils sont utilisés en accord avec les Appenders afin d'associer la cible de l'enregistrement avec la manière de tracer les données. Log4j fournit plusieurs Layouts comme :

- org.apache.log4j.SimpleLayout : qui permet de journaliser de façon simple les événements.
- org.apache.log4j.PatternLayout : qui permet de journaliser les messages en fonction d'un modele ou motif.
- org.apache.log4j.HTMLLayout : qui permet de journaliser les événements au format HTML. Chaque journalisation produit un document HTML complet.
- org.apache.log4j.XMLLayout : qui permet de journaliser les événements au format XML en conjugaison avec un FileAppenders.
- org.apache.log4j.net.SMTPAppender : qui permet de journaliser les événements en les envoyant par email.

Configuration dynamique

La configuration précédente avec les Appenders dans les fichiers sources n'est pas très pratique et mélange le code source avec des éléments de configuration de la journalisation. De même, lors du développement, tous les messages seront affichés avec un niveau WARN par exemple et en production, seuls les messages de type ERROR devront être tracés.
Avec l'API Log4j, il existe trois méthodes pour configurer les Loggers. La première est la configuration par défaut que nous venons d'aborder avec une gestion dans les sources. La seconde configuration est réalisée à l'aide d'un fichier de propriétés avec le format habituel clé=valeur.
Enfin le dernier type de configuration repose sur un fichier au format XML.

Nous allons utiliser la solution intéressante proposée à partir d'un fichier de propriétés. Dans un tel fichier, chaque Logger peut être configuré et les Appenders et les layouts peuvent y être paramétrés. Nous pouvons ainsi aisément modifier le formatage des messages. Dans un fichier de configuration, chaque Appender doit avoir un nom afin de pouvoir y faire référence lors de la configuration des Loggers. Les Appenders sont préfixés par log4j.appender

La déclaration d'un Appender est réalisée sous la forme suivante :
log4j.appender.NomAppender=ClasseAppender

Le paramètre NomAppender est le nom que nous souhaitons utiliser pour notre Appender et le paramètre ClasseAppender est la classe d'implémentation de l'Appender. Voici un exemple de configuration d'un Appender simple pour la console :

```
#CONSOLE est l'Appender de type ConsoleAppender
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.SimpleLayout
```

L'appender est correctement configuré, nous devons maintenant paramétrer les Loggers pour qu'ils utilisent les Appenders. La configuration des Loggers est similaire à celle des Appenders, la forme est la suivante :

```
log4j.logger.nomDuLogger=niveau, appender1, appender2, …
```

Le paramètre nomDuLogger est le nom du Logger indiqué dans l'instruction Logger.getLogger(nomDuLogger) et représente une structure pointée, identique à la notion de paquetage. Le paramètre niveau est le nom du niveau attribué au logger (ERROR, DEBUG, …) . S'il n'est pas précisé, le niveau est hérité du nœud parent ou positionné à DEBUG. Le paramètre appenderX est le nom d'un Appender déclaré dans le fichier.

Pour mettre en place ce mécanisme nous allons créer un fichier nommé log4j.properties dans le paquetage ressources.

```
#définition du niveau et des Appender du rootLogger
#(ordre : DEBUG – INFO – WARN – ERROR – FATAL )
log4j.rootLogger=ERROR, CONSOLE
#CONSOLE est l'Appender de type console
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
#définition du format des messages 2005-16-18 14:53:37 DEBUG [Main] Hello World
log4j.appender.CONSOLE.layout.ConversionPattern=%d %-5p %c - %F:%L - %m%n
```

Nous allons donc pouvoir intercepter les traces de types ERROR et donc FATAL, car celles-ci sont plus basses dans la hiérarchie. Ce type de trace est associé à l'Appender nommé CONSOLE qui utilise la classe org.apache.log4j.ConsoleAppender et qui va donc afficher les traces dans la console JAVA avec le format défini par le Layout ou modèle.

```
Public class Exemple {

   private static final Logger logger = Logger.getLogger(getClass()) ;

   public Exemple(){
     logger.error(''Hello Log'') ;
     logger.log(Level.FATAL, ''Hello Log Constructeur'') ;
   }

}
```

Maintenant pour que ce service soit opérationnel et que les Loggers déclarés dans les fichiers sources utilisent ce fichier de propriétés, il est nécessaire de le mettre en place dans une classe chargée au démarrage. Pour cela, nous ajoutons un attribut dans le fichier de configuration de l'application web.xml Enfin, nous terminons le paramétrage en précisant le fichier de configuration dans la classe de gestion des plug-ins ou tout autre classe lancée au démarrage de l'application.

```
<?xml version=''1.0'' encoding=''UTF-8''?>
<web-app>
   <!-- le chemin pour acceder au fichier de propriétés log4j - →
   <context-param>
      <param-name>log4jfichierConfig</param-name>
      <param-value>WEB-INF/.../log4j.properties</param-value>
   </context-param>
. . .
```

Ensuite, cela dépend de l'application JEE, du framework utilisé (Struts, Spring, etc ..)