Journalisation des évènements avec l'API Logging de Java

Hugo Etiévant

Dernière mise à jour : 18 avril 2004

Qu’est-ce que la journalisation ?

La journalisation consiste à garder les traces sur un support sûr des évènements survenus dans un système ou dans une application.

Un ou plusieurs fichiers de log au format prédéfini sont générés en cours d’exécution et conservent des messages informant sur la date et l’heure de l’événement, la nature de l’événement et sa gravité par un code ou une description sémantique, éventuellement d’autres informations : utilisateur, classe, etc…

Pourquoi créer un fichier de log ?

Les journaux peuvent utilement être réutilisés par :

- un administrateur afin de produire des statistiques sur l'utilisation d'un système (par exemple les logs du serveur web Apache)
- un développeur afin de détecter des défaillances et de corriger les bugs qui en sont responsables, il est plus facile de repérer la source d'une défaillance si le journal est dense en informations (fonctions appelées, valeurs des paramètres passés…)
- un développeur pour éviter de polluer son code avec des println()
- un utilisateur peut utiliser un journal afin de revenir sur un crash et refaire les opérations qui n'auraient été perdues (transactions)

L'API Logging de Java

L'API utilisée : « java.util.logging » est fournie par défaut dans le JDK 1.4.

D'usage simple, elle permet de journaliser des évènements dans un fichier au format texte ou XML. Différents niveaux de sévérité sont applicables aux messages journalisés. Et la fonction standard se décline en une myriade de fonctions spécifiques.

Cette API doit être appelée dans toute classe qui nécessite de journaliser des informations :

```
import java.util.logging.*;
```

Création d'un journal (1)

Pour utiliser un journal dans une classe, il faut :

- créer un attribut statique et protégé faisant référence au journal
- appeler la méthode « getLogger() » de la classe « Logger » de l'API « Logging » qui prend en paramètre le nom du journal
- l'affecter à un flux de sortie, généralement un fichier

Par convention, on donne au journal le nom complet de la classe en cours avec la hiérarchie des packages. Ainsi, chaque classe aura son journal. On peut aussi donner le nom du package pour que toutes les classes utilisent le même journal. Mais en toute rigueur, vous pouvez donner le nom que vous souhaitez au journal.

Exemple :

```
protected static Logger logger =
        Logger.getLogger("myPackage.mySubPackage.myClasse");
```

Création d'un journal (2)

Le journal n'est créé qu'une seule fois, si plusieurs classes appellent la méthode « getLogger() » avec le même nom, le journal sera créé au premier appel ; ensuite, le journal sera récupéré mais pas recréé aux prochains appels. C'est l'aspect statique du journal.

Il faut ensuite créer un pointeur vers un fichier, ce pointeur est une instance de la classe « FileHandler » de l'API Logging.

Puis on l'associe au journal via la méthode « addHandler() » du journal à laquelle on passe le pointeur en paramètre.

Exemple :

```
Handler fh = new FileHandler("myLog.log");
logger.addHandler(fh);
```

Création d’un journal (3)

A noter que le (ou les) fichier(s) de log se distingue(nt) du journal. Les premiers sont la représentation physique du journal. Alors que le journal « Logger » est le système logiciel de gestion des messages du journal.

Fichier de log (1)

Le fichier stockant le journal peut se voir affecter des propriétés particulières :

- le nom du fichier peut contenir des caractères spéciaux définissant un motif « pattern »
- une taille limite pour le fichier « limit » (exprimée en octets, infini par défaut)
- un nombre de fichiers cycliques « count » (1 par défaut)
- un mode d'appel « append » (true ou false)

Syntaxes :

FileHandler()

FileHandler(String pattern)

FileHandler(String pattern, boolean append)

FileHandler(String pattern, int limit, int count)

FileHandler(String pattern, int limit, int count, boolean append)

Fichier de log (2)

Le système choisit lui-même un nom de fichier :

```
Handler fh = new FileHandler();
```

Le fichier portera le nom indiqué :

```
Handler fh = new FileHandler("myLog.log");
```

Le fichier est recrée (false) ou repris tel quel (true) :

```
Handler fh = new FileHandler("myLog.log", false);
```

Le journal sera divisé en 5 fichiers de 10000 octets chacun. Leurs nom sera en myLog.log.i avec i de 0 à 4 (motif par défaut) :

```
Handler fh = new FileHandler("myLog.log", 10000, 5);
```

```
Handler fh = new FileHandler("myLog.log", 10000, 5, false);
```

Fichier de log (3)

Le motif du nom de fichier peut être défini par l'utilisateur à l'aide des caractères spéciaux suivants :

Caractère	Description
/	séparateur de répertoires dans le système de fichier local
%t	répertoire temporaire du système
%h	répertoire de connexion de l'utilisateur (équivalent de « user.home »)
%g	le nombre généré automatiquement par la rotation cyclique des fichiers
%u	un nombre aléatoire unique

Pour déspécialiser le caractère « % », il faut le doubler.

Exemple :

```
Handler fh = new FileHandler("%t/myApps.%g.log", 10000, 4);
```

Message à journaliser

Pour poster un message « msg » dans le journal, il faut utiliser la fonction « log() » de l'objet « Logger ». L'argument « level » définit le niveau de criticité du message « msg » passé en paramètre. Si ce niveau est l'un de ceux gérés par le journal, alors, le message sera redirigé vers tous les flux de sortie associés au journal.

Syntaxe :

```
void log(Level level, String msg)
```

Exemple :

```
logger.log(Level.WARNING, "argument out of limit");
```

Cet exemple envoie le message "argument out of limit" de niveau « Level.WARNING » au journal.

Niveaux de criticité (1)

La classe « Level » définit 7+2 niveaux de criticité pour les messages à journaliser. Ces niveaux ont chacun un poids différent. Le tableau ci-dessous présente les niveaux du plus fort au plus faible.

Niveau	Description
ALL	Tous les niveaux
SEVERE	Niveau le plus élevé
WARNING	Avertissement
INFO	Information
CONFIG	Configuration
FINE	Niveau faible
FINER	Niveau encore plus faible
FINEST	Niveau le plus faible
OFF	Aucun niveau

Niveaux de criticité (2)

Par défaut, un journal ne transfère au flux de fichier que les messages de niveaux supérieur ou égale à « INFO », les autres sont ignorés.

Cela peut aisément être changé on modifiant grâce à la méthode « setLevel() » le niveau critique à partir duquel on commence à transférer au fichier de log les messages reçus par le journal.

Syntaxe :

```
void setLevel(Level newLevel)
```

Exemple pour logger tous les messages :

```
logger.setLevel(Level.ALL);
```

Niveaux de criticité (3)

Voici quelques méthodes de l'objet « Level » utiles pour la création de filtres et de formateurs (voir plus loin) :

Méthode	Description
getName()	Retourne le nom du niveau
getLocalizedName()	Retourne le nom du niveau convertit dans la langue configurée dans la JVM
intValue()	Retourne la valeur entière codant le niveau
parse(String name)	Retourne un objet « Level » associé au nom passé en paramètre

Alias de log()

Pour simplifier l'envoie de messages au journal, des alias de la fonction « log() » ont été crés : il n'est plus nécessaire de spécifier le niveau de criticité car les alias portent pour identificateur le nom de ces niveaux.

Syntaxes :

void **severe**(String msg)
void **warning**(String msg)
void **info**(String msg)
void **config**(String msg)
void **fine**(String msg)
void **finer**(String msg)
void **finest**(String msg)

Exemple :

```
logger.severe("bad template definition");
```

est équivalent à :

```
logger.log(Level.SEVERE, "bad template definition");
```

Autres fonctions de log (1) – log()

Il existe de multiples déclinaisons de la fonction « log() ».

Fonction habituelle :
void **log**(Level level, String msg)

Rajout d'un argument :
void **log**(Level level, String msg, Object param1)

Rajout d'un tableau d'arguments :
void **log**(Level level, String msg, Object[] params)

Rajout d'une exception :
void **log**(Level level, String msg, Throwable thrown)

Passage d'un enregistrement « LogRecord » :
void **log**(LogRecord record)

Autres fonctions de log (2) – logp()

La fonction « log() » se décline en une famille de méthodes plus précises sur l'origine de la journalisation : les méthodes « logp() ».

Spécification du niveau, du nom de la classe et de la méthode depuis on émet un message :

void **logp**(Level level, String sourceClass, String sourceMethod, String msg)

Spécification du premier paramètre passé à la méthode depuis laquelle on journalise :

void **logp**(Level level, String sourceClass, String sourceMethod, String msg, Object param1)

Spécification d'un tableau d'arguments passés à la méthode journalisée :

void **logp**(Level level, String sourceClass, String sourceMethod, String msg, Object**[]** params)

Spécification de l'exception à journaliser :

void **logp**(Level level, String sourceClass, String sourceMethod, String msg, Throwable thrown)

Autres fonctions de log (3) – logrb()

Famille de méthodes plus précises spécifiant un fichier « bundleName » de localisation du message.

Spécification du niveau, du nom de la classe et de la méthode depuis on émet un message ainsi que le nom du fichier de localisation :

void **logrb**(Level level, String sourceClass, String sourceMethod, String bundleName, String msg)

Spécification du premier paramètre passé à la méthode depuis laquelle on journalise :

void **logrb**(Level level, String sourceClass, String sourceMethod, String bundleName, String msg, Object param1)

Spécification d'un tableau d'arguments passés à la méthode journalisée :

void **logrb**(Level level, String sourceClass, String sourceMethod, String bundleName, String msg, Object**[]** params)

Spécification de l'exception à journaliser :

void **logrb**(Level level, String sourceClass, String sourceMethod, String bundleName, String msg, Throwable thrown)

Autres fonctions de log (4) – entering

Cette famille de méthodes permet de journaliser l'entrée dans une méthode.

Spécification du nom de la classe et de la méthode qui est démarrée:

void **entering**(String sourceClass, String sourceMethod)

Spécification du nom de la classe et de la méthode qui est démarrée ainsi que du premier argument passé à cette méthode :

void **entering**(String sourceClass, String sourceMethod, Object param1)

Spécification du nom de la classe et de la méthode qui est démarrée ainsi que d'un tableau des arguments passés à cette méthode :

void **entering**(String sourceClass, String sourceMethod, Object[] params)

Autres fonctions de log (5) – exiting

Cette famille permet de journaliser la sortie d'une méthode.

Spécification du nom de la classe et de la méthode de laquelle on s'apprête à sortir :

void **exiting**(String sourceClass, String sourceMethod)

Spécification du nom de la classe et de la méthode de laquelle on s'apprête à sortir ainsi que de l'objet retourné:

void **exiting**(String sourceClass, String sourceMethod, Object result)

Autres fonctions de log (6) – throwing

Cette méthode permet de journaliser le fait de la levée d'une exception.

Spécification du nom de la classe et de la méthode depuis lesquelles est levée l'exception elle aussi spécifiée:

void **throwing**(String sourceClass, String sourceMethod, Throwable thrown)

Exemple :

```
try {…}
catch (Exception e) {
        logger.throwing("BankImpl", "findBranch", e);
}
```

LogManager

Les différent journaux utilisés dans les applications tournant dans une JVM donnée sont gérés de façon transparente par le « LogManager ».

L'objet journal « Logger » envoie au « LogManager » un « LogRecord » contenant le message et le niveau spécifié mais aussi d'autres informations implicites d'après l'analyse de la pile mémoire des appels de méthode : nom de classe et de la méthode (et ses arguments) depuis laquelle est envoyé le message à journalisé, mais aussi la date et heure, le numéro de thread…

Tout « LogRecord » passe par un filtre « Filter » afin de savoir si le message peut être journalisé ou pas, entre autre par évaluation du niveau « Level ». Il passe également par un formateur « Formatter » qui est associé au pointeur « Handler » vers le flux de sortie et qui en transforme la représentation (texte, XML…).

Formats de sortie (1)

Les pointeurs « Handler » permettent d'associer un flux de sortie au journal. Celui utilisé tout au long des exemples de ce cours est le pointeur vers fichier « FileHandler », mais il en existe d'autres :

Handler	Description
ConsoleHandler	correspond au flux d'erreur standard « System.err »
FileHandler	simple fichier texte du système de fichiers
SocketHandler	flux réseau vers une machine et un port à déterminer
MemoryHandler	buffer cyclique en mémoire vive
StreamHandler	flux quelconque de sortie

Note : tout flux de sortie doit être fermé par sa méthode « close() » après utilisation.

Formats de sortie (2)

Syntaxes :

ConsoleHandler()

FileHandler()
FileHandler(String pattern)
FileHandler(String pattern, boolean append)
FileHandler(String pattern, int limit, int count)
FileHandler(String pattern, int limit, int count, boolean append)

SocketHandler()
SocketHandler(String host, int port)

MemoryHandler()
MemoryHandler(Handler target, int size, Level pushLevel)

StreamHandler()
StreamHandler(OutputStream out, Formatter formatter)

Formats de sortie (3)

Il est possible au développeur de créer son propre pointeur vers un nouveau flux en étendant la classe « Handler ».

Filtrage (1)

Le filtrage permet au développeur de créer un filtre, c'est-à-dire une classe qui implémente l'interface « Filter », qui défini la fonction suivante :

public boolean isLoggable(LogRecord record)

qui retourne « true » si le journal doit envoyé le « LogRecord » au flux de sortie, « false » sinon.

Un filtre est associé à un journal via la méthode « setFilter() » de l'objet « Logger ».

Syntaxe :

void setFilter(Filter newFilter)

Exemple :

logger.setFilter(new myFilter());

Filtrage (2)

Exemple de filtre :

```
import java.util.logging.*;
import java.lang.*;
// filtre
public class myFilter implements Filter {
        public myFilter() {
                super();
        }
        // décide si l'enregistrement doit être publié ou pas
        public boolean isLoggable(LogRecord record) {
                return true;
        }
}
```

Formatage (1)

Il existe deux classes de formatage de base dans l'API Logging :

Formatter	Description
SimpleFormatter	Formatage en texte simple
XMLFormatter	Format XML

Il est là encore possible, d'en créer un nouveau en étendant la classe « Formatter ».

Un formateur s'applique à un flux de sortie par la méthode « setFormatter() » de l'objet « Handler ».

Syntaxe :

```
void setFormatter(Formatter newFormatter)
```

Exemple :

```
fh.setFormatter(new myFormatter());
```

Formatage (2) – format texte

Exemple :

April 18, 2004 11:45:05 PM myApps main
SEVERE: bad template syntaxe

April 18, 2004 11:45:05 PM myApps myFct
WARNING: second argument forgotten

April 18, 2004 11:45:05 PM myApps main
FINER: 35 blocks found

Formatage (3) – format XML

Exemple de sortie XML :

```
<?xml version="1.0" encoding="windows-1252" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
  <date>2004-02-04T23:24:19</date>
  <millis>1075933459913</millis>
  <sequence>0</sequence>
  <logger>myExample</logger>
  <level>INFO</level>
  <class>myExample</class>
  <method>main</method>
  <thread>10</thread>
  <message>my info</message>
</record>
</log>
```

Formatage (4) – format XML

Tag	Description
Log	Racine du document XML
Record	Définit un enregistrement (une ligne) du journal
Date	Date et heure de journalisation
Millis	Timestamp unix de la date et heure
Sequence	Numéro d'ordre du fichier stockant le journal si cycle
Logger	Nom du système de log ayant réalisé la journalisation
Level	Niveau de criticité du message journalisé
Class	Nom de la classe
Method	Nom de la méthode
Thread	Numéro de processus
Message	Texte du message

Formatage (5)

Exemple de formateur :

```
import java.util.logging.*;

// formateur
class myFormatter extends Formatter {
        // formatage d'un enregistrement
        public String format(LogRecord record) {
                StringBuffer s = new StringBuffer(1000);
                s.append("<tr><td>" + record.getLevel() + "</td>");
                s.append("<td>" + formatMessage(reccord) + "</td></tr>\n");
                return s.toString();
        }
        // entête du fichier de log
        public String getHead(Handler h) {
                return "<html>\n<body>\n<table>\n";
        }
        // fin du fichier de log
        public String getTail(Handler h) {
                return "</table>\n</body>\n</html>\n";
        }
}
```

Formatage (6)

Voici les méthodes à étendre du « Formatter » :

Méthode	Description
String format(LogRecord record)	Formatage d'un enregistrement
String getHead(Handler h)	Ecriture d'un entête au journal
String getTail(Handler h)	Ecriture d'une fin au journal

Voici une méthode propre au système, sensible à la configuration de la JVM :

Méthode	Description
String formatMessage(LogRecord record)	Formatage par application des méthodes de localisation des types de date et d'heure du message

LogRecord

Un enregistrement « LogRecord » du journal contient des données accessibles pour les besoins des filtres et formateurs. En voici les accesseurs :

Méthode	Description
getLevel()	Niveau « Level » de l'enregistrement
getLoggerName()	Nom du journal
getMessage()	Message à journalisé
getMillis()	Timestamp unix au format long
getParameters()	Tableau (Object[]) des paramètres passés à la méthode journalisée
getSequenceNumber()	Numéro d'ordre de l'enregistrement
getSourceClassName()	Nom de la classe d'appel de la fonction log()
getSourceMethodName()	Nom de la méthode d'appel de la fonction log()
getThreadID()	Numéro du thread d'appel de la fonction log()
getThrown()	Exception (Throwable) associée au journal

Fichier de configuration des journaux

Les propriétés par défaut du « LogManager » peuvent être changées dans le fichier de configuration suivant : « lib/logging.properties » du répertoire du JRE de la JVM.

Extrait du fichier de configuration :

```
handlers=java.util.logging.FileHandler, java.util.logging.ConsoleHandler
.level= INFO
java.util.logging.FileHandler.pattern = %h/java%u.log
java.util.logging.FileHandler.limit = 50000
java.util.logging.FileHandler.count = 1
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter
java.util.logging.ConsoleHandler.level = INFO
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter
myPackage.mySubPackage.myClass.level = SEVERE
```

Pour appliquer un autre fichier de configuration à la compilation, il faut utiliser la commande suivante :

```
java -Djava.util.logging.config.file= myFile
```

Conlusion

Cette API fourni un moyen efficace et flexible de journaliser les événements d'une application Java.

De plus, les journaux sont sérialisables permettant de journaliser des applications distribuées.

Historique

- **18 avril 2004** : création du document (37 diapos)

Agissez sur la qualité de ce document en envoyant vos critiques et suggestions à l'auteur.

Pour toute question technique, se reporter au forum Java de Developpez.com

Hugo Etiévant
cyberzoide@yahoo.fr
http://cyberzoide.developpez.com/