Ingénierie des architectures logicielles : technologies et méthodes
Rappels théoriques : les threads et les sockets

Version 1.2

Vincent Englebert - Fabian Gilson

6 février 2013

Table des matières

Historique

1.0 Agrégation des anciens documents sur les threads et les sockets.

1.1 Remise en forme et ajout d'exemples.

1.2 Ajout des exemples et conf Log4j.

1 Les threads

1.1 Définition d'un thread

Le terme *thread* est utilisé pour "thread of control", et signifie littéralement "fil de contrôle". Le fil de contrôle d'un programme est la suite des instructions qui sont effectivement exécutées. Il est en quelque sorte le chemin emprunté dans le programme par une exécution. A l'exécution d'un processus, plusieurs threads peuvent coexister en parallèle.

Threads et processus sont intrinsèquement liés. Les processus peuvent être vus comme des threads au niveau de l'OS. Chaque processus dispose de son propre espace mémoire. Un processus peut contenir plusieurs threads, chacun de ces threads utilisant l'espace mémoire du processus. Néanmoins, chacun de ces threads peut disposer de variables locales.

Par exemple, la machine virtuelle Java constitue un processus OS contenant plusieurs threads (*garbage collector*, *AWT threads*, *main classes*, etc.).

1.2 Implémentation des threads en Java

Java permet de programmer des threads de manière relativement aisée (par rapport à C par exemple), grâce aux classes `Thread` (par héritage) et `Runnable` (par délégation).

1.2.1 Par délégation

On crée une classe qui implémente l'interface `Runnable` contenant une seule méthode `run()`. Il s'agit ensuite d'instancier cette classe et d'initialiser un thread avec l'objet (constructeur) obtenu avant de le lancer avec la méthode `start`.

```
public class MyRunnable implements Runnable {
  public void run() { /* running code */}
  public static void main (String[] args) {
    Thread t = new Thread(new MyRunnable());
    t.start();
  }
}
```

1.2.2 Par héritage

On crée une classe qui hérite de la classe `Thread` et qui implémente la méthode `run()`. Cette classe implémente elle-même la classe `Runnable`.

```
public class MyThread extends Thread {
  public MyThread() { /* constructor */ }
  public void run() { /* running code */ }
  public static void main(String[] args) {
    MyThread t1 = new MyThread();
    MyThread t2 = new MyThread();
    t1.start();
    t2.start();
  }
}
```

Nous utiliserons la première méthode, car elle présente l'avantage de permettre la programmation de threads sans devoir hériter de la classe Thread, ce qui préserve la capacité d'hériter d'une autre classe, Java ne permettant que l'héritage simple.

1.3 Cycle de vie d'un Thread

A tout moment de sa "vie", un thread peut être dans l'un des états suivants :

Created le thread est instancié, mais sa méthode `start()` n'a pas encore été invoquée.

Running la méthode `start()` a été appelée et le thread occupe actuellement le processeur.

Runnable la méthode `start()` a été appelée, mais le thread n'occupe pas le processeur (en attente).

Non-runnable le thread n'est plus en compétition pour l'obtention du processeur (attend une ressource, par exemple).

Terminated le thread est terminé et a relâché ses ressources.

Java fournit différentes méthodes pour gérer ce cycle de vie :

`start()` fait passer le thread de l'état *created* à *runnable* ou *running* ;

`yield()` force le thread à abandonner le processeur. En d'autres termes, cette méthode force le thread à passer de l'état *running* à *runnable* ;

`suspend()` fait passer le thread à l'état *Non-runnable* ;

`resume()` fait passer le thread de l'état *Non-runnable* à *Runnable* ;

`stop()` fait passer le thread dans l'état *Terminated*.

`join()` bloque le thread courant jusqu'à la terminaison du thread sur lequel la méthode est invoquée ;

Attention, notez bien que `suspend()`, `resume()` et `stop()` sont *dépréciées*. Leur usage est **fortement** déconseillé.

1.4 Concurrence et partage

1.4.1 Type de conflits

Lors de l'accès concurrent à une ressource partagée, deux types de conflits peuvent survenir :

read/write si un thread lit une ressource en même temps qu'un autre écrit sur celle-ci, la donnée lue peut être incohérente.

write/write écriture simultanée de deux threads sur une même ressource. Plus particulièrement, ce conflit se manifeste lorsque chacun des threads lit une donnée et la modifie par la suite simultanément, l'un anéantissant l'effet de l'autre.

1.4.2 Synchronisation (exclusion mutuelle)

Pour éviter ces conflits, on peut recourir à des verrous *mutex* (contraction de *mutually exclusive*). Ceci se fait en Java en déclarant une méthode ou une partie de code `synchronized`. Un tel bloc de code implique que le thread qui l'appelle possède le verrou sur la classe correspondante (il n'y a qu'un seul verrou par objet). Ainsi, dans la classe du *snippet* suivant, la méthode `A` ne peut pas être invoquée en parallèle par deux ou plusieurs threads au sein d'un même objet. Le thread invoquant la méthode `A` possède le verrou sur cette classe. Donc, la méthode `B` ne pourra pas être invoquée non plus. Par contre, `C` n'est pas synchronisée, donc aucun verrou n'est nécessaire.

```
public class Test {
  public synchronized void A(){ /* some code */ };
  public synchronized void B(){ /* some other code */ };
  public void C(){ /* more code */ };
}
```

1.4.3 Synchronisation conditionelle

Les méthodes `notify`, `notifyAll` et `wait` vues au cours permettent d'élaborer un mode de communication entre threads.

`wait()` suspend l'exécution du thread cible et lui fait relâcher le verrou qu'il détient sur l'objet courant (objet de la méthode courante, c-à-d. `this`).

`notify()` réveille un thread (qui avait appelé la méthode `wait()`). Attention, il ne sera effectivement réveillé que s'il obtient le verrou sur l'objet (s'il en existe un).

`notifyAll()` fonctionne comme la méthode `notify()`, sauf qu'elle réveille tous les threads en attente [1].

1. Une explication plus détaillée en anglais venant de http://www.javamex.com/tutorials/notifyall.shtml. ≪ The `notify()` method is generally used for resource pools, where there are an arbitrary number of "consumers" or "workers" that take resources, but when a resource is added to the pool, only one of the waiting consumers or workers can deal with it. The `notifyAll()` method is actually used in most other cases. Strictly, it is required to notify waiters of a condition that could allow multiple waiters to proceed. But this is often difficult to know. So as a general rule, if you have no particular logic for using `notify()`, then you should probably use `notifyAll()`, because it is often difficult to know exactly what threads will be waiting on a particular object and why. ≫

1.5 Deadlock

Il y a *deadlock*, ou inter-blocage, lorsque deux threads ou plus se mettent dans l'attente de la libération de deux verrous ou plus, et que les conditions sont telles que ces deux verrous ne seront jamais libérés. Il existe **4 conditions nécessaires et suffisantes** pour qu'il y ait inter-blocage :

1. les threads impliqués accèdent à des ressources ayant un accès mutuellement exclusif ;
2. l'acquisition des ressources est incrémentale. En d'autres termes, un thread garde les ressources qui lui ont déjà été allouées quand il est en attente de l'acquisition d'autres ressources ;
3. une fois qu'un thread a obtenu une ressource, on **ne** peut **pas** la lui retirer de force. La libération de ressources par un thread ne se fait que de manière volontaire par celui-ci ; et
4. il existe un cycle d'attente entre threads, en ce sens que chaque thread détient une ressource que son suivant attend d'acquérir.

Pour qu'il n'y ait pas deadlock, il suffit donc que l'une de ces conditions ne soit pas remplie. Voici un exemple de deadlock.

```
public class DeadlockSample {

  public static void main(String[] args) {

    final Object resource1 = "resource1";
    final Object resource2 = "resource2";

    // t1 tries to lock resource1 then resource2
    Thread t1 = new Thread() {

      @Override
      public void run() {

        synchronized (resource1) {
          System.out.println("Thread 1: locked resource 1");

          try {
            Thread.sleep(50);
          } catch (InterruptedException e) {}

          synchronized (resource2) {
            System.out.println("Thread 1: locked resource 2");
          }
        }
      }
    };

    // t2 tries to lock resource2 then resource1
    Thread t2 = new Thread() {

      @Override
      public void run() {

        synchronized (resource2) {
          System.out.println("Thread 2: locked resource 2");

          try {
            Thread.sleep(50);
          } catch (InterruptedException e) {}

          synchronized (resource1) {
            System.out.println("Thread 2: locked resource 1");
          }
        }
      }
    };

    // If all goes as planned, deadlock will occur, and the program will never exit.
    t1.start();
    t2.start();
  }
}
```

1.6 Safety et Liveness

Deux types de propriétés sont intéressantes dans le cas des programmes concurrents : les *safety properties* (propriétés de sécurité) et les *liveness properties* (propriétés de vivacité). Intuitivement, les premières établissent que *rien de mal ne peut se passer*, les secondes établissent qu'*éventuellement quelque chose de bien finira par se produire*. Par exemple, l'exclusion mutuelle est une propriété de sécurité, qui permet aux ressources partagées de ne pas se trouver dans un état incohérent. La prévention du deadlock peut être aussi vue en partie comme une propriété de sécurité, dans le sens où le deadlock est un état incohérent. Néanmoins, la prévention du deadlock est également une propriété de vivacité, car lorsqu'il y a deadlock, plus rien de bon ne peut arriver.

Une autre propriété de vivacité est la propriété de *progress*, qui établit que dans n'importe quel état du système, une action donnée pourra éventuellement être exécutée. L'inverse de cette propriété est ce que l'on appelle un état de *starvation*, dans lequel, par exemple, un processus monopolise une ressource de telle manière qu'un autre processus ne pourra jamais exécuter certaines actions. Un exemple de starvation est illustré dans le *snippet* suivant.

```
public class Starvation implements Runnable {
  private final Object resource_;
  private final String message_;
  private final boolean fair_;

  // program entry point
  public static void main(String[] args) {

    boolean fair = false;
    if (args != null && args.length >= 1 && args[0].equals("fair")) {
      fair = true;
    }

    // get the number of available cpus; do twice as much threads
    final int runners = Runtime.getRuntime().availableProcessors() * 2;
    System.out.println("starting " + runners + " runners");

    final Object resource = new Object();

    // create sample runners and start them
    for (int i = 1; i <= runners; i++) {
      (new Thread(new Starvation(resource, String.valueOf(i), fair))).start();
    }

    // suspend main thread
    synchronized (Starvation.class) {
      try {
        Starvation.class.wait();
      } catch (InterruptedException ignored) {}
    }
  }

  public Starvation(Object resource, String message, boolean fair) {
    resource_ = resource;
    message_ = message;
    fair_ = fair;
  }

  public void run() {
    synchronized (this) {
      for (;;) {
        synchronized (resource_) {
          print(message_);
          // some delay; if we keep synchronized on the contended resource,
          // scheduling isn't fair, possibly leading to thread starvation
          try {
            (fair_ ? resource_ : this).wait(100);
          } catch (InterruptedException ignored) {}
        }
      }
    }
  }

  private static void print(String s) {
    synchronized (System.out) {
      System.out.print(s);
    }
  }
}
```

1.7 Exercices

1.7.1 Synchronisation de thread

Créez deux threads et attendez qu'ils se terminent. Généralisez à N threads.

1.7.2 Le verrou Mutex

Implémentez une classe qui joue le rôle d'un verrou mutex en Java. Vous aurez certainement besoin d'utiliser des méthodes `synchronized`, ainsi que les méthodes `wait()`, `notify()` et `notifyAll()`.

1.7.3 Tower Bridge

On vous demande de simuler en Java l'accès concurrent par des voitures à un pont à une seule voie. Ainsi, les voitures ne peuvent avoir accès de manière concurrente au pont que si elles roulent dans le même sens. Le programme Java consistera donc en une classe représentant le pont, et deux classes représentant respectivement les voitures venant de la gauche et celles venant de la droite. Mais avant toute chose, essayez de bien identifier quels peuvent être les problèmes de *safety* et de *liveness* dans ce problème.

1.7.4 Power Bridge

Un pont supporte une charge maximale de 30 tonnes. Ce pont est traversé par des camions dont le poids est de 15 tonnes ainsi que par des voitures dont le poids est de 5 tonnes. On vous demande de gérer l'accès au pont de sorte que :

1. La charge maximale du pont soit respectée.
2. la priorité soit donnée aux camions : lorsqu'une voiture et un camion demandent l'accès au pont, le camion doit être choisi en priorité, sous réserve que la capacité maximale du pont soit respectée.

Écrire un programme qui simule les règles de partage du pont ci-dessus. Votre programme modélisera les camions et voitures sous la forme de threads.

1.7.5 Robots

Un certain nombre de robots se trouvent sur une grille, et se déplacent de manière aléatoire et instantanée. Modélisez ce problème au moyen d'une classe `Grille` qui joue le rôle de la ressource) et d'une classe `Robot` jouant le rôle du thread). Nous faisons l'hypothèse que lorsqu'un robot décide de se rendre à une position, il n'abandonne pas sa décision. Quel type de problème peut survenir ici ? Illustrez. Comment le résoudre ?

1.7.6 Pipe-Line

On vous demande d'écrire un programme qui convertit des nombres en base 10 vers la base 2. Pour ce faire, on vous demande d'utiliser un *pipe-line* de threads. Votre programme doit être constitué de 8 threads :

- Le premier thread lit depuis le clavier le nombre à convertir, puis, transfère cette information vers le thread suivant.
- Six threads affichent 0 ou 1 selon que le nombre reçu par le thread soit divisible ou non par deux, et transfèrent le nombre divisé par 2 au thread suivant (sauf si le nombre divisé est égal à zéro).
- Le dernier thread du pipe-line doit afficher une erreur si le nombre à convertir dépasse la capacité du pipe-line.

Tel que le pipe-line est décrit ici, le résultat de la conversion doit normalement afficher le nombre en base 2 à l'envers.

1.7.7 Argument et synchronisation

Lors d'un appel de méthode synchronisée, l'évaluation des arguments de la méthode est-elle réalisée avant ou après l'obtention du verrou ? Que dit la documentation de Java sur ce point ? Ecrivez un programme Java qui met en évidence la sémantique déduite du premier point.

1.7.8 Tri Fusion multithreadé

Implémentez un tri par fusion avec plusieurs threads. Le tri par fusion consiste à scinder récursivement le tableau en deux parties, de trier chacune de ces parties, et ensuite de les interclasser. Le but ici sera de faire le tri des deux parties du tableau de manière parallèle.

Pour ce faire, vous implémenterez la classe `Runnable TableSort`, qui hérite de la classe `MyTable`, dont la définition suit :

```
public class MyTable {
  public int[] tab;
  public int size;

  public MyTable(int[] t, int size) { /* Constructor */ }

  public void sort() {
    /* sort tab[0..size/2] et tab[(size/2)+1..size] and let remaining of table unchanged */
  }
}
```

1.7.9 Producteur / Consommateur

On désire développer une petite application simulant les comportements concurrents d'un ensemble de producteurs et de consommateurs de messages. D'un côté, les producteurs produisent des messages qu'ils stockent dans un buffer commun ; de l'autre, les consommateurs récupèrent dans ce même buffer les messages (dans l'ordre où ils y ont été placés).

On doit respecter un certain nombre de contraintes et pouvoir paramétrer l'application :

1. un producteur ne peut produire un message que si le buffer n'est pas plein et, bien sûr, un consommateur ne peut lire un message que si le buffer n'est pas vide ;
2. le débit de chaque producteur et de chaque consommateur est paramétrable à sa construction (il représente le temps d'attente entre deux productions ou deux consommations).

Ecrire les classes Producteur et Consommateur permettant de mettre en oeuvre ces spécifications. On pourra implanter le buffer avec une `ArrayList`. Chaque producteur et consommateur sera exécuté par un thread différent, en concurrence avec les autres.

1.7.10 Le dîner des Philosophes

Cinq philosophes partagent une table circulaire, où ils possèdent chacun une chaise. Un philosophe passe sa vie alternativement à penser et à manger. Au milieu de cette table se trouve un plat de pâtes et un philosophe a besoin de deux baguettes pour pouvoir manger. Mais malheureusement, les philosophes ne disposent que de 5 baguettes au total. Une baguette est placée entre chaque philosophe et ceux-ci ne peuvent utiliser que les baguettes qui se trouvent immédiatement à leur gauche et à leur droite. De plus, tous les philosophes agissent de la même manière : ils s'assoient, ils prennent la baguette gauche, ensuite la droite, mangent, reposent la baguette gauche, reposent la baguette droite, et philosophent.

Modélisez ce problème grâce à un programme java, dans lequel les fourchettes seront les ressources et les philosophes les threads. Montrez un scénario d'exécution dans lequel il y a deadlock. Proposez une modification de cet énoncé afin de supprimer le deadlock.

2 Les Sockets Java

Les *sockets* Java sont les interfaces permettant de communiquer avec des ordinateurs distants via les protocoles TCP (*Transmission Control Protocol*) et UDP (*User Datagram Protocol*).

2.1 *Transmission Control Protocol*

La programmation réseau par TCP se fait en Java via les classes `Socket`, et `ServerSocket`. La classe `Socket` est utilisée du côté *client* (entité qui initie la connexion) et la classe `ServerSocket` est utilisée du côté *serveur* recevant les demandes de connexion des clients.

2.1.1 Côté serveur

Tout d'abord, on instancie la classe `ServerSocket` en lui fournissant en paramètre le port sur lequel elle devra écouter. Ensuite, on accepte la connexion du client, via la méthode `accept()` de la classe `ServerSocket`. Cette méthode retourne une instance de la classe `Socket` et représente la connexion vers le client. Pour récupérer les flux de lecture et d'écriture du socket on utilise les méthodes `getInputStream()` et `getOutputStream()` de la classe `Socket`.

```
// create listening socket
ServerSocket s = new ServerSocket(5000);
// start listening for incoming requests
Socket clientConn = s.accept();
// retrieve incoming and ougoing streams on socket
InputStream in = clientConn.getInputStream();
OutputStream out = clientConn.getOutputStream();
```

2.1.2 Côté client

On instancie la classe `Socket` en lui fournissant l'adresse (instance de la classe `InetAddress`) du serveur et le port sur lequel il écoute. Comme du côté serveur, on récupère les flux de lecture et d'écriture du socket via les méthodes `getInputStream()` et `getOutputStream()` de la classe `Socket`.

```
// open socket to server with address (url) "the.server.address"
InetAddress addr = InetAddress.getByName("the.server.address");
Socket s = new Socket(addr,5000);
// retrieve incoming and ougoing streams on socket
InputStream in = clientConn.getInputStream();
OutputStream out = clientConn.getOutputStream();
```

2.2 *User Datagram Protocol*

2.2.1 UDP monocast

Le protocole UDP est un protocole orienté message (TCP étant orienté connexion) non fiable, c'est-à-dire, **(i)** il n'y pas de garanties sur la réception du message, **(ii)** ni sur le respect de l'ordre et **(iii)** des messages peuvent être répliqués.

Par contre, l'intégrité du message est garantie. Les systèmes d'exploitation imposent généralement une limite de 8Kb à la taille du message envoyé. Si cette dernière est trop contraignante, le système d'exploitation peut être configuré pour augmenter cette limite sans toutefois pouvoir dépasser une limite absolue de 64Kb.

En Java, l'utilisation du protocole réseau UDP se fait via les classes `DatagramPacket` pour les paquets eux-mêmes et `DatagramSocket` qui permet d'envoyer et de recevoir ces paquets. Côté client, on commence par instancier la classe `DatagramSocket` en spécifiant éventuellement le port UDP. Ensuite, on emballe les données à envoyer dans un paquet de type `DatagramPacket`. Le constructeur prend en argument un tableau d'octets (`byte[]`), la longueur de ce tableau (`int`), l'adresse IP du destinataire (`InetAddress`), et son port (`int`).

```
// create socket
DatagramSocket udpSocket = new DatagramSocket(5000);
// create address of remote host with address "the.server.address"
InetAddress addr = InetAddress.getByName("the.server.address");
```

```
// build datagram
String buffer = "Hello World, it's " + new Date();
DatagramPacket p = new DatagramPacket(buffer,buffer.length,addr,5000);
// send datagram on socket
udpSocket.send(p);
```

Côté serveur, la méthode `receive()` de la classe `DatagramSocket` permet d'attendre et de recevoir le paquet émis (de type `DatagramPacket`).

```
DatagramPacket p = udpSocket.receive();
```

2.2.2 UDP multicast

L'UDP *multicast* permet d'envoyer des messages, non plus à un seul destinataire, mais à un ensemble de destinataires qui se sont inscrits dans un *groupe*. L'envoi de messages se fait vers une adresse *multicast* allant de 224.0.0.0 à 239.255.255.255 (définie par le protocole UDP, adresses dites de classe D). Concrètement, seules les adresses dans le range 225.0.0.0 → 238.255.255.255 sont utilisables pour les applications utilisateur, les autres adresses étant réservées [2]. Comme pour UDP monocast, des paquets peuvent être perdus, répliqués, délivrés dans le mauvais ordre et les destinataires d'un paquet ne verront pas tous le même schéma de perte ou d'ordre d'arrivée. Si le protocole est suffisamment fiable dans la majorité des cas, ces limites doivent être prises en compte. En particulier, UDP multicast est connu pour ses pertes importantes lorsque de nombreux clients émettent en même temps de nombreux paquets.

Voici comment on utilise du multicast en Java. On instancie la classe `MulticastSocket` avec le numéro du port d'écoute. Pour rejoindre un groupe de récepteurs identifié par une adresse IP multicast, il faut utiliser la méthode `MulticastSocket.joinGroup()`. Cette adresse est globale à l'infrastructure réseau selon la configuration. L'envoi et la réception de données se fait de manière analogue à celle d'UDP monocast (en utilisant la classe `MulticastSocket` plutôt que `DatagramSocket`.

```
// create socket
MulticastSocket ms = new MulticastSocket(5000);
// create address
InetAddress sessAddr = InetAddress.getByName("239.0.0.1");
// join multicast group
ms.joinGroup(sessAddr);
```

2.3 Exercices

2.3.1 Salut, Bonjour

Écrivez un programme Java qui répond "Salut !" quand on lui dit "Bonjour !". Faites le d'abord en UDP et ensuite en TCP.

2.3.2 Il est minuit Docteur ?

Un ensemble de récepteurs est intéressé de recevoir régulièrement l'heure d'un processus émetteur. Écrivez les classes `Emetteur` et `Recepteur`.

Remarques

Les exceptions ont été délaissées dans les exemples afin de garder les *snippets* compréhensibles. Nous laissons au lecteur le soin de compléter le code.

Remerciements

A collaboré à ce document : M. François Vermaut.

2. cfr http://www.iana.org/assignments/multicast-addresses

A Exemple d'utilisation de `log4j`

Afin de debugger facilement, il est utile d'imprimer le maximum d'information quelque part. La librairie `Log4j` [3] d'*Apache* permet de logger facilement des messages vers divers supports (console, fichier et socket TCP). Un exemple est donné dans le *snippet* suivant.

```
import org.apache.log4j.Logger;

import be.fundp.info.util.logging.MyLogger;

public class Log4JSample {

  private static final Logger logger = MyLogger.getLogger(Class.class);

  public static void main(String[] args) {

    // init logger
    MyLogger.initLogger("log4j.properties");
    logger.info("logger started");

    try {
      logger.debug("enter tricky part");
      // do something tricky
    } catch (Exception e) {
      // do something to correct problem
      logger.warn("something went wrong, but necessary actions were taken", e);
    }

    try {
      // do something
    } catch (Exception e) {
      // cannot take actions, but can continue
      logger.error("something went wrong and cannot take necessary corrective actions", e);
    }

    try {
      // do something that cannot crash
    } catch (Exception e) {
      logger.fatal("something bad happened, cannot continue", e);
      System.exit(1);
    }
  }
}
```

3. http://logging.apache.org

B Exemple de fichier de configuration de `log4j`

L'avantage de `log4j` est qu'il est facilement configurable via un fichier `properties` ou `xml`. Un exemple vous est donné dans le listing suivant.

```
###
#
# Log4j sample property file.
#
# Contains properties for a console appender, socket appender (for Chainsaw,
# cfr http://logging.apache.org/chainsaw/index.html) and a rolling file appender.
#
# Shows also how to define log level per classes.
#
# @author fgi
#
###

# declare all appenders, printing ALL levels of logging info
log4j.rootLogger= ALL,consoleAppender, socketAppender, fileAppender

# console appender definition
log4j.appender.consoleAppender=org.apache.log4j.ConsoleAppender
log4j.appender.consoleAppender.layout=org.apache.log4j.PatternLayout
log4j.appender.consoleAppender.layout.ConversionPattern= [%d] [%C{1}.%M:%L] - %m%n

# socketAppender for basic use of chainsaw via a socket
log4j.appender.socketAppender=org.apache.log4j.net.SocketAppender
log4j.appender.socketAppender.RemoteHost=localhost
log4j.appender.socketAppender.Port=4445
log4j.appender.socketAppender.LocationInfo=false
log4j.appender.socketappender.layout=org.apache.log4j.PatternLayout
log4j.appender.socketAppender.layout.ConversionPattern= [%d] [%t @ %C.%M] - %m%n

# The fileAppender's type specified as RollingFileAppender, i.e. log output appended to a file
.
log4j.appender.fileAppender=org.apache.log4j.RollingFileAppender
log4j.appender.fileAppender.File=_logs//log
log4j.appender.fileAppender.MaxFileSize=5MB
log4j.appender.fileAppender.MaxBackupIndex=2
log4j.appender.fileAppender.immediateFlush=true
log4j.appender.fileAppender.append=true
log4j.appender.fileAppender.Encoding=UTF-8

# The fileAppender is assigned a layout PatternLayout described right behind.
log4j.appender.fileAppender.layout=org.apache.log4j.PatternLayout
log4j.appender.fileAppender.layout.ConversionPattern= [%d] [%t] [%C %M] [%F:%L] - %m%n
# corresponding chainsaw pattern for logfilereciever: [TIMESTAMP] [THREAD] [CLASS METHOD] [
    FILE:LINE] - MESSAGE

# log levels per class (can be of type DEBUG, INFO, WARN, ERROR or FATAL)
log4j.logger.mypackagepath.MyClass=WARN
```