

Monitoring et optimisation JBoss 7.x, Jboss EAP 6.x

David THIBAU – 2017

david.thibau@gmail.com



Agenda

- **Monitoring et tuning Java**
 - Concepts fondamentaux de Java
 - Java et la mémoire
 - Pools et collectes
 - Tuning serveurs
 - Outils de commande en ligne pour le monitoring
 - Particularités JavaEE
 - Pools et Caches dans JBoss
- **Outils de monitoring JBoss**
 - La console d'administration
 - Le système de trace
 - CLI
- **Tiers Web**
 - Configuration du sous-système
 - Tuning session
 - Pool de threads
- **Tiers métier**
 - Tuning des pools EJBs
 - Cache stateful
- **Tiers persistance**
 - Configuration data-source
 - Pool de connexions
 - 2nd Level Cache



Concepts fondamentaux de Java



JVM

La JVM est l'environnement d'exécution pour les applications Java

Elle permet notamment :

- l'interprétation du byte code

- l'interaction avec le système d'exploitation

- la gestion de sa mémoire grâce au ramasse miettes

Son mode de fonctionnement est relativement similaire à celui d'un ordinateur : elle exécute des instructions qui manipulent différentes zones de mémoire dédiées de la JVM.

Une application Java ne fait pas d'appels directs au système d'exploitation (sauf en cas d'utilisation de JNI)



Gestion de la mémoire

Pour faciliter la gestion de la mémoire, Java propose les mécanismes suivants :

- Il n'est pas possible d'allouer de la mémoire explicitement : c'est la création d'un nouvel objet avec l'opérateur *new* qui alloue la mémoire requise

- La JVM dispose d'un ramasse miettes qui se charge de libérer la mémoire des objets inutilisés

Comme conséquences :

- il n'est pas possible de connaître le moment où la mémoire d'un objet sera libérée

- le ramasse miettes ne dispense pas le développeur de connaître son mode de fonctionnement et de prendre quelques précautions pour éviter les fuites de mémoires

Une JVM 32bits utilise un adressage sur 32 bits ce qui lui permet de gérer jusqu'à 4 Go de mémoire.



Zones mémoire

Plusieurs zones de mémoire sont utilisées par la JVM :

- les **registres** (*register*) : ces zones de mémoires sont utilisées par la JVM exclusivement lors de l'exécution des instructions du byte code.

- les **pires** (*stack*) dédiées aux threads

- un **tas** (*heap*) pour les objets

- une **zone de méthodes** (*method area*) stockant les définitions de classes, le code des méthodes et les variables *static*. Elle est partagée par toutes les threads

- Le **code cache** stockant les méthodes compilées (taille par défaut de 32Mo)



Stack

La pile d'une thread contient les variables qui ne sont accessibles que par la thread (variables locales, paramètres, valeurs de retour des méthodes).

- Seules des données de type primitif et des références à des objets sont stockées dans la pile. Elle ne peut pas contenir d'objets.
- La taille d'une pile peut être précisée à la machine virtuelle. - **Xss**
- Si la taille d'une pile est trop petite pour les besoins des traitements d'un thread alors une exception de type **StackOverflowError** est levée.
- Si la mémoire de la JVM ne permet pas l'allocation de la pile d'un nouveau thread alors une exception de type **OutOfMemoryError** est levée.



Le tas

Cette zone de mémoire est partagée par tous les threads de la JVM : elle stocke toutes les instances des objets créés.

- Tous les objets et tableaux créés sont obligatoirement stockés dans le tas (**heap**).
- La libération de cet espace mémoire est effectuée par le ramasse miettes (**garbage collector**).
- La taille du tas peut être fixe ou variable durant l'exécution de la JVM : dans ce dernier cas, une taille initiale est fournie et cette taille peut grossir jusqu'à un maximum défini.
- Si la taille du heap ne permet pas le stockage d'un objet en cours de création, alors une exception de type *OutOfMemoryException* est levée.



Cycle de vie des classes

Une classe ou une interface suit le cycle de vie suivant :

1. chargement (loading)
2. liaison (linking)
3. initialisation (initialization)
4. instanciation (instantiation)
5. récupération de la mémoire (garbage collection)
6. finalisation (finalization)
7. déchargement (unloading)



Chargement des classes

Un ***classloader*** est un objet qui charge dynamiquement et initialise des classes Java il hérite de la classe *java.lang.ClassLoader*.

Pour charger une classe, le chargeur de classe effectue généralement plusieurs opérations :

- Vérification si la classe est déjà chargée et initialisée

- Tentative de chargement du *bytecode*

- Si le chargement réussi, initialisation du *bytecode* dans la JVM



Java standard et Serveurs

Java EE

Un programme Java standard utilise typiquement la stratégie de recherche d'une classe suivante :

- Recherche dans la distribution Java (**bootstrap** classes de *rt.jar*)
- Si pas trouvé, recherche dans les **extensions** : jars présent dans `$JRE_HOME/lib/ext`
- Si pas trouve, recherche dans les répertoires/archives indiqués par la variable d'environnement **CLASSPATH**

Un serveur Java EE implémente sa propre stratégie de chargement de classe car il doit faire coexister dans la même application Java plusieurs unités de déploiements indépendantes



Bytecode

Le **bytecode** est un langage intermédiaire entre le code source et le code machine

- La JVM fournit un environnement d'exécution pour le *bytecode* en le convertissant en code machine du système d'exploitation utilisé
- Le *bytecode* peut être modifié avant son exécution par un *classloader* dédié. (AOP, Profiling par exemple)
- Le compilateur transforme le code source Java en fichiers **.class** contenant entre autre le *bytecode*.



Jeu d'instructions

Les instructions de la JVM sont des opérations basiques qui combinées permettent de réaliser les traitements.

Une instruction est composée d'un **code opération** (*opcode*) suivi d'aucune, une ou plusieurs **opérandes** qui représentent les paramètres de l'instruction.



Désassembleur *javap*

Il est possible d'avoir une version lisible d'un fichier *.class* en utilisant l'outil de désassemblage ***javap***

Utilisé sans options, *javap* affiche le package, les champs et méthodes publiques ou protégées d'une classe

Avec l'option **-c**, on obtient les instructions du bytecode

Son utilisation peut être intéressante pour l'optimisation d'algorithme ou pour observer les différences entre compilateurs



Example

Compiled from "ClasseDeTest.java"

```
public class org.formation.test.ClasseDeTest extends java.lang.Object{  
public org.formation.test.ClasseDeTest();
```

Code:

```
0:   aload_0  
1:   invokespecial   #8; //Method java/lang/Object."<init>":()V  
4:   return
```

```
public static void main(java.lang.String[]);
```

Code:

```
0:   iconst_1  
1:   istore_1  
2:   goto          8  
5:   iinc          1, 1  
8:   iload_1  
9:   bipush 10  
11:  if_icmple      5  
14:  return
```




Compilateur JIT

Pour éviter l'interprétation des méthodes à chaque appel, la JVM peut utiliser un compilateur à la volée

Le JIT compile en code natif le byte code d'une méthode, stocke le résultat de cette compilation et exécute ce code compilé chaque fois que la méthode est invoquée

La machine virtuelle proposée par Sun peut fonctionner selon deux modes.

- Dans le mode *client*, c'est la réduction du temps de compilation qui est privilégiée au détriment des optimisations.
- Dans le mode *serveur*, c'est l'optimisation qui est privilégiée ce qui allonge le temps de compilation.



Options de la JVM

La JVM Hotpsot possède des options standards et des options non standards qui peuvent être dépendantes de la plate-forme d'exécution.

Les paramètres standards sont préfixés par **-X**

`java -X` permet d'obtenir un résumé des options standard supportées par la JVM.

Les options **-XX** sont elles supportées exclusivement par la machine virtuelle HotSpot, quelques unes sont également supportés par la version OpenJDK



Options standard (1)

- Xint** : Désactive le compilateur JIT
- Xbatch** : Désactive la compilation en tâche de fond
- Xbootclasspath:bootclasspath** : Définit les répertoires, les jar ou les archives zip qui composent les classes de bootstrap.
- Xbootclasspath/a:path** : Ajoute des répertoires, des jar ou des archives zip aux classes de bootstrap
- Xcheck:jni** : Effectue des contrôles poussés sur les paramètres utilisés lors d'appels à des méthodes natives avec JNI. Si un problème est détecté lors de ces contrôles, alors la machine virtuelle est arrêtée avec une erreur fatale.
- Xnoclassgc** : Désactiver la récupération de la mémoire par le ramasse miettes des classes chargées mais inutilisées. Ceci peut légèrement améliorer les performances mais provoquer un manque de mémoire.



Options standard (2)

- Xincgc** : Active les collectes incrémentales pour le ramasse miettes. Ceci permet de réduire les longs temps de pauses nécessaires au ramasse miettes en réalisant une partie de son activité de façon concomitante avec l'exécution de l'application.
- Xloggc:file** : Active les traces d'exécution du ramasse miettes dans un fichier de log fourni en paramètre. Cette option est prioritaire sur l'option `-verbose:gc` si les deux sont fournies à la JVM
- Xms<n>** : Permet de préciser la taille initiale du tas.
- Xmx<n>** : Permet de préciser la taille maximale du tas.
- Xprof** : Active l'affichage sur la console de traces de profiling.
- Xss<n>** : Permet de définir la taille de la pile des threads



Java et la mémoire

Pools et collecte générationnelle
Algorithmes du GC
Tuning



Problèmes liés à la performance

Problème	Conséquences	Causes possibles
Mémoire : La JVM ne peut pas se contenter de la mémoire qu'on lui a allouée	L'application CRASH ou est inutilisable	<ul style="list-style-type: none">- Mauvais dimensionnement des ressources- Mauvaise configuration JVM- Fuites mémoire dans le code applicatif
CPU : Le pourcentage de CPU reste élevé pendant longtemps	Les performances se dégradent, certaines interactions peuvent être sans réponses	Principalement applicatif
I/O : L'application effectue des nombreuses lecture/écriture	Les performances sont mauvaises	<ul style="list-style-type: none">- Due à la nature de l'application- Configuration des traces



Pools et Collecte générationnelle



Mémoire Heap et Permanent

Java 7

La JVM gère 2 types de mémoire créée dès le démarrage :

La **Heap** correspond à la zone dédiée aux allocations des objets Java.

Sa taille peut être fixe ou variable (-Xms, -Xmx).

La mémoire **Permanent** stocke les informations statiques (~fichiers .class).

Sa taille peut être fixe ou variable. (--XMaxPermSize)



Heap et MetaSpace

Java 8

Java 8 élimine la zone Permanent et la remplace par la **MetaSpace**:

Les classes sont dorénavant stockées dans la mémoire **native**.

Par défaut, la *MetaSpace* n'est pas limitée (limite = RAM disponible)

Il est cependant possible d'indiquer des flags permettant de fixer une valeur :

- Initiale : `-XX:MetaspaceSize`
- Maximale : `-XX:MaxMetaspaceSize`



Pools de la Heap

La HotSpot Java VM, propose pour la heap les pools mémoire suivants :

Eden Space : La zone d'allocation initiale des objets

Survivor Space : Contient les objets ayant survécu à une collecte de la zone Eden.

Tenured Generation : Contient les objets qui ont existé un certain temps dans la zone *survivor*.



Génération et collecte

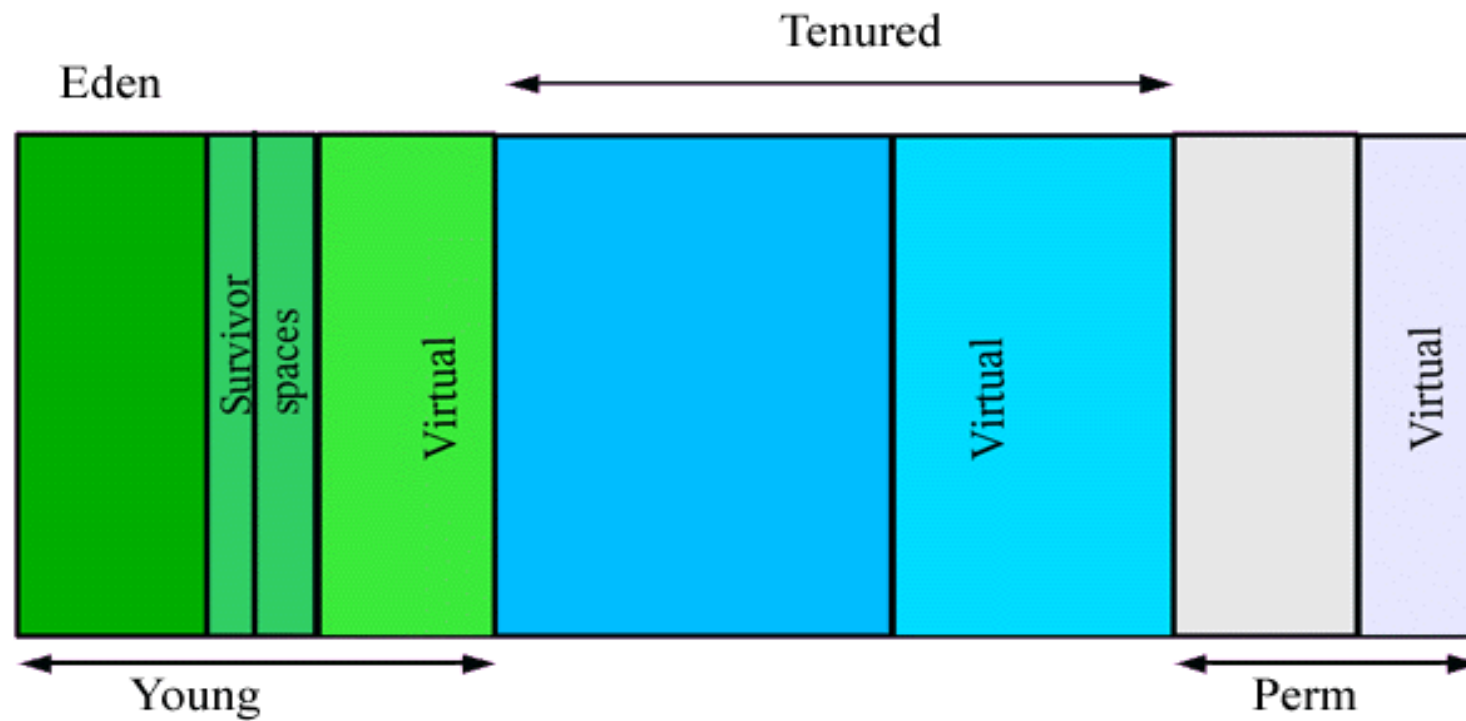
La JVM HotSpot définit 2 générations d'objets :

La **jeune génération** contient 3 pools : l'Eden space, et 2 espaces pour les survivants

La **vielle génération** contient 1 pool : *tenured*

- ✓ Tous les objets commencent par l'*Eden Space* (et la plupart meurent dans ce pool)
- ✓ Lorsque la JVM effectue une collecte sur l'Eden space, les objets vivants sont déplacés dans un des espaces pour survivants
- ✓ Au bout d'un certain temps, la JVM déplace les survivants encore référencés vers la vielle génération
- ✓ Lorsque la vielle génération atteint sa limite, une collecte complète est effectuée. Cette collecte a des impacts sur l'application car elle concerne l'ensemble des objets présents en mémoire

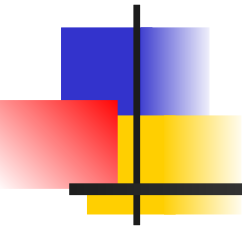
Pools générationnels





Types de collecte

- Les collectes sur l'espace jeune dénommées **collecte partielle ou mineure** (*partial/minor collection*) se produisent fréquemment mais sont rapides car cette zone reste petite.
 - Les objets qui survivent à un certain nombre de collecte jeunes sont éventuellement **promus** dans l'espace dédié à la vielle génération
- Les collectes sur l'espace vieux, dénommées **collecte complète ou majeure** (*full/major collection*) prennent plus de temps. Elles doivent être rares



Algorithmes de collecte de la HotSpot

- Collecte série
- Collecte parallèle
- Collecte parallèle avec compactage
- Collecte CMS (Concurrent Mark-Sweep)
- Garbage first : G1



Série/parallèle – Pauses de l'application

Plusieurs choix de design sont possibles pour l'algorithme de collecte :

Série ou parallèle : Avec l'option parallèle, les tâches de collecte peuvent être réparties sur différents CPUs.

La parallélisation ajoute de la complexité et favorise la fragmentation

Arrêt ou non de l'application : L'exécution de l'application peut être stoppée lors de la collecte. Avec l'option parallèle, l'application n'est stoppée que pendant de courtes pauses.




Compactage / Non compactage / Copie

Compactage : Après la collecte d'espace, le ramasse miettes déplace les objets vivants dans des zones contiguës.

Il est alors plus facile d'allouer des objets au premier emplacement libre

Non compactage : Le ramasse miettes libère la place des objets morts sans déplacements. La collecte est plus rapide mais génère de la fragmentation qui pénalise les allocations futures

Copie : Les objets vivants sont copiés dans une autre zone mémoire, la zone source est entièrement libérée et devient très rapide pour de nouvelles allocations. L'inconvénient est le temps de copie



Métriques de performance pour le gc

Différents métriques sont utilisés pour mesurer la performance de la collecte :

- **Débit** : Le pourcentage de temps non passé dans la collecte sur une longue période
- **Surcharge** : L'inverse du débit. Le pourcentage de temps passé par la collecte.
- **Temps de pause** : Le temps cumulé où l'application est stoppée
- **Fréquence** : La fréquence des collectes comparée à l'exécution de l'application
- **Empreinte** : La taille de la mémoire occupée
- **Promptitude** : Le délai entre la mort d'un objet et la libération de la mémoire



Type d'applications

- Une application interactive nécessitera des temps de pause bas
- Une application sans interaction est plus intéressée par un bon débit.
- Une application temps-réel nécessitera des temps de collecte garantis et faibles.
- L'empreinte sera le métrique le plus important pour des systèmes embarqués



Types de collecte

La JVM HotSpot dispose de plusieurs algorithmes de collecte :

Collecte série : Adaptée lorsqu'un seul CPU est disponible, la collecte provoque l'arrêt de l'application

Collecte parallèle : Profite des machines multi-CPU, la collecte s'effectue en utilisant plusieurs CPU

Collecte parallèle avec compactage : Profite des machines multi-CPU et compacte la zone

Collecte CMS (Concurrent Mark-Sweep) :
Algorithme minimisant les temps de pause

G1 (Garbage first) : Contrôle du temps de pause et compactage (Late JDK7 et JDK8). A vocation d'améliorer CMS



Utilisation de la collecte série

La collecte série est adaptée au machine de type client (~64 Mo de heap) qui ne nécessite pas de faible temps de pause

A partir de Java5, cet algorithme est automatiquement choisi pour des machines restreint à 64Mo de heap

On peut forcer l'utilisation de cet algorithme en utilisant l'option :
-XX:+UseSerialGC



Utilisation de la collecte parallèle

La collecte parallèle est adaptée aux applications s'exécutant sur plusieurs CPUs et qui n'ont pas de contraintes au niveau des temps de pause.
(Exemple : Traitements batch)

Les collectes de la vieille génération continuent de se produire

Avec Java5, cet algorithme est automatiquement sélectionné pour les machines de type serveur (plus de 2 CPUs et 2Go de Heap)

L'utilisation de cet algorithme peut être forcée avec l'option :

`-XX:+UseParallelGC`



Utilisation de la collecte parallèle avec compactage

Ce type d'algorithme adapté aux applications bénéficiant de plusieurs CPUs, permet des allocations plus rapides

Il n'est pas adapté aux machines partagés par plusieurs applications (la collecte peut monopoliser la totalité des CPUs) ou alors diminuer le nombre de threads utilisées pour la collecte avec l'option
`-XX:ParallelGCThreads=n`

Avec Java6, cet algorithme est automatiquement sélectionné pour les machines serveur

Cet algorithme peut être explicitement spécifié via l'option :
`-XX:+UseParallelOldGC`



Utilisation de la collecte CMS

Typiquement, les applications qui ont plus de 2 CPUs et qui utilisent des objets avec de longs temps de vie. (Par exemple les serveurs web)

Cet algorithme peut également être adapté aux applications nécessitant de faibles temps de pause, s'exécutant sur un seul processeur mais ayant une taille de vieille génération modeste.

Pour utiliser la collecte CMS :

- XX:+UseConcMarkSweepGC

En mode incrémental :

- XX:+CMSIncrementalMode

Enfin avec Java6, la collecte CMS peut être forcée pour les appels à *System.gc()* et *System.getRuntime().gc()* avec :

- XX:+ExplicitGCInvokesConcurrent



Limitation CMS

- Pas super performant pour les grosses tailles de Heap ($Xmx > 6\text{ GB}$)
- A cause de la fragmentation, les pauses GC pauses peuvent augmenter jusqu'à des niveaux inacceptables
- Les temps de pause du GC sont imprévisibles



Apport de G1

Développé pour remplacer CMS.

- Apparu depuis JDK 7 update 4.
- C'est un algorithme qui compacte
- On peut spécifier les temps de pause désiré (-
XX:MaxGCPauseMillis)

La heap est divisée en régions de taille uniforme. (entre 1 et 32Mo en fonction de la taille de la heap ou fixé explicitement via *-XX:G1HeapRegionSize=<size in Mb>*)

Certains ensemble de région se voient attribuer les mêmes rôles que les zones eden, survivor, old des autres algorithmes mais il n'y a plus de taille fixe



Principes

G1 effectue une phase de marquage concurrente sur toutes les régions pour déterminer les objets vivants

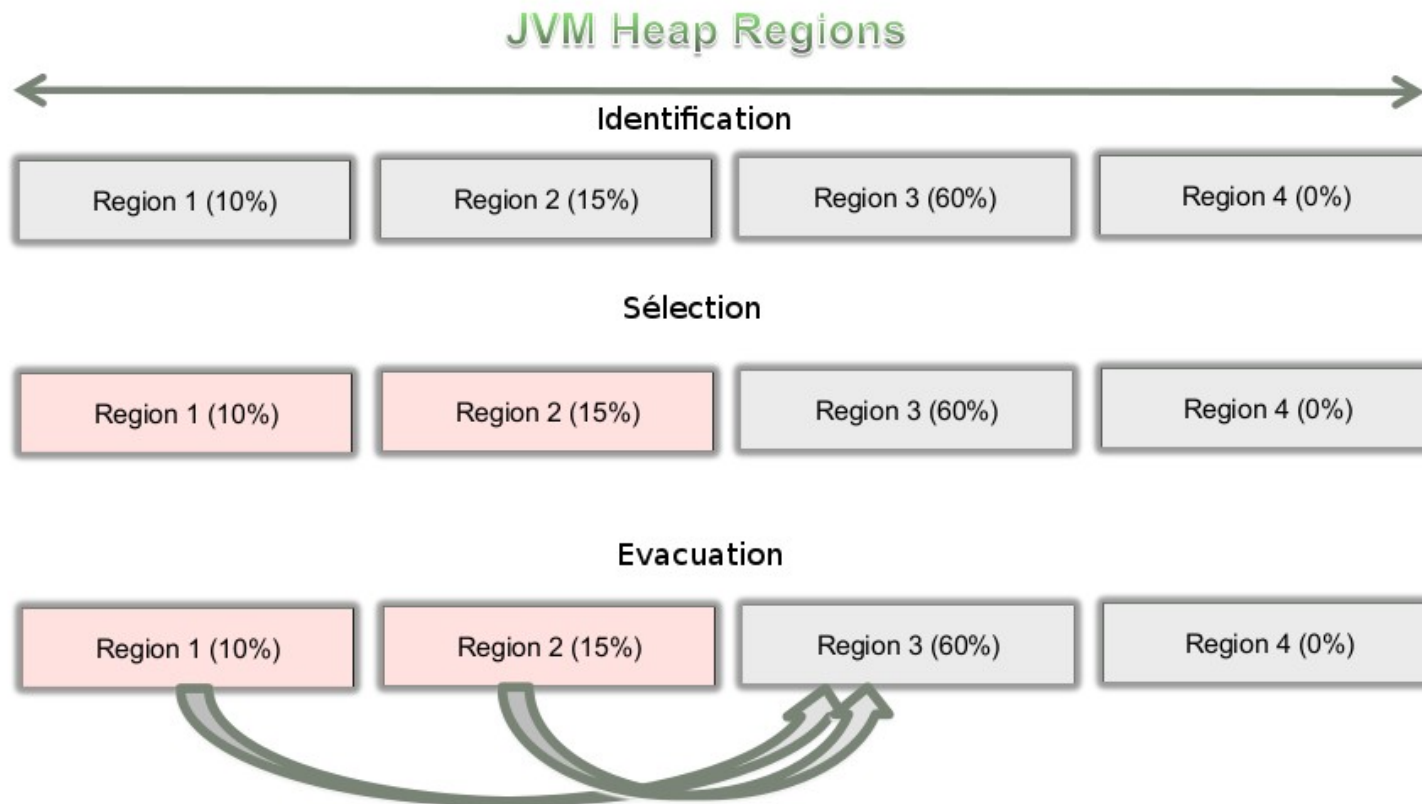
G1 concentre ses collectes et compactage sur les zones de la heap permettant de récupérer le plus d'espace (d'où le nom Garbage First)

G1 sélectionne le nombre de région à collecter en fonction des objectifs de temps de pause.

La collecte sur les zones est effectuée par **évacuation**, i.e. Copie des objets vivants vers une zone vierge => compactage.

Durant l'évacuation, les threads applicatives sont stoppées

Illustration



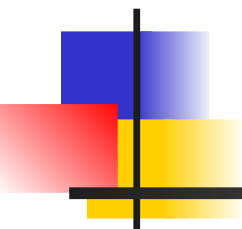


Cas d'usage

G1 est surtout destiné au serveur applicatif qui ont beaucoup d'objets vivants et une implémentation multi-threaded qui s'exécute sur des machines avec plusieurs CPUs

On peut envisager de migrer vers G1 lorsque :

- La durée ou la fréquence des Full GC sont trop importantes
- Le taux d'allocation ou de promotion d'objet varie considérablement
- Des temps de pauses trop long non voulus



Tuning serveur



Introduction

Les particularités d'une application serveur est qu'elle doit répondre le plus rapidement possible à ses requêtes clients

=> Utilisation importante des CPUs pour un temps limité (réponse)

Les nombreuses options de la JVM permettent d'améliorer cet objectif



Classe de machines

La JVM distingue 2 types de machines (*client* et *server*)

- En fonction du type de machine, elle utilise des mécanismes (JIT compiler et code cache) et des valeurs par défaut différents (Taille de Heap, collecte, etc..)
- La JVM détecte automatiquement le type de machine en fonction de la RAM et du nombre de CPUs
- Il est cependant possible et conseillé de spécifier le type de machine au démarrage.

Pour un serveur, l'option est **-server**




Dimensionnement explicite

Pour tous les pools, la JVM propose une valeur initiale et une valeur maximale.

Si ces valeurs sont différentes, cela peut provoquer des redimensionnements de zone durant l'exécution et donc pénaliser les performances.

Il est donc conseillé de fixer la valeur initiale à la valeur maximale



Paramètres de dimensionnement

Pour la taille totale de la heap : **-Xms** et **-Xmx**

Pour la taille de l'emplacement des classes :

- Java 7 : **-XX:PermSize** et **-XX:MaxPermSize**
- Java 8 : **-XX:MaxMetaspaceSize** (Juste limiter la taille)

Taille de la jeune génération : **-Xmn**

Taille des Survivants : **-XX:SurvivorRatio**



Espace total de la heap

Comme les collectes surviennent lorsque les générations sont remplies, le débit de l'application dépend principalement de l'espace mémoire disponible

=> La **mémoire totale est le facteur le plus influent sur la performance des collectes.**

Pour trouver la valeur minimale, il faut estimer l'espace requis par l'application en rythme de croisière
=> Mesures de l'empreinte mémoire sous une charge normale pour une application Web

- Pour trouver la valeur à pas dépasser, il faut mesurer les temps de pause sous une collecte majeure

Typiquement, pour les serveurs JavaEE :

- Pour les machines 32 bits 2Go
- Pour les machines 64 bits 4Go (valeur moyenne)



Jeune génération

Le second facteur influent est la proportion de mémoire affectée à la jeune génération.

- Plus la jeune génération est importante moins il y a de collectes mineures.
- Pour un total borné, plus vous augmentez la jeune génération moins il reste de place pour la vieille génération ... ce qui augmente la fréquence des collectes majeures.

Le choix optimal dépend en fait de la distribution du temps des vies des objets de votre application.

Le but étant de minimiser le temps cumulé des collectes.
=> Encore une fois, seules les mesures peuvent apporter une réponse



Espace des survivants

Le paramètre ***SurvivorRatio*** peut être utilisé pour fixer la taille des pools survivor mais n'influe pas énormément sur les performances

Par exemple, `-XX:SurvivorRatio=6` indique un ratio de 1/6ème pour chaque espace des survivants

Si les espaces des survivants sont trop petits, les objets seront copiés directement dans l'espace de la vieille génération; si ils sont trop grands, ils seront inutilement vides.

A chaque collecte, la JVM détermine un nombre seuil de survie produisant la copie d'un objet vers la vieille génération. Ce seuil est choisi de sorte à garder l'espace des survivants à moitié rempli.

L'option `-XX:+PrintTenuringDistribution`, peut être utilisée pour afficher la valeur du seuil et l'âge des objets dans la jeune génération.



Recommandations

Après avoir fixé l'espace totale de la mémoire, il faut répartir le total de la heap entre les différentes générations. .

=> Allouer le maximum de mémoire à la jeune génération, tant que cela ne provoque pas trop de collecte de la vieille génération ou de trop long temps de pause

=> Pour une collecte en série, la jeune génération ne doit pas dépasser la moitié de la taille totale de la heap



OutOfMemoryError

Si la JVM a un problème de mémoire, elle lance une *OutOfMemoryError*

Analyser le détail de l'erreur pour déduire la marche à suivre

Java heap space

GC Overhead limit exceeded

PermGen space / Metaspace

Requested array size exceeds VM limit

request size bytes for reason. Out of swap space?

D'autre part l'option

-XX:+HeapDumpOnOutOfMemoryError permet de générer un dump mémoire pour analyse lorsque provient le *OutOfMemory*



Java heap space

Ce type d'erreur indique qu'un objet n'a pas pu être alloué dans la heap. Elle peut provenir :

D'une erreur de configuration, *-Xmx* est trop faible pour l'application

Un problème applicatif : fuite mémoire ou mauvaise utilisation des finalizers

=> Augmenter *Xms* et *Xmx*, si le problème persiste, retour aux développeurs



GC Overhead limit exceeded

Cela indique que le processus Java passe la plupart de son temps à effectuer des collectes

=> Augmenter *Xms* et *Xmx*



Requested array size exceeds VM limit

Ce type d'erreur indique que
l'application a essayé d'allouer un
tableau plus grand que la taille de la
heap.

Soit la taille de la heap est trop faible

Soit il y a un bug dans l'application



PermGen Space

Java 7

Ce type d'erreur indique que le pool de la génération permanente est rempli.

Cela peut arriver lorsque l'application (ou le framework utilisé) utilise un grand nombre de classes

Il faut alors modifier l'option
-XX:MaxPermSize=n




Metaspace

Java 8

Ce type d'erreur indique que la mémoire native pour le chargement des classes et des méta-données est insuffisant

- Si *MaxMetaSpaceSize* est précisé, l'augmenter
- Sinon, diminuer Xms et Xmx
- Sinon, achetez de la RAM



request size bytes for reason. Out of swap space?

Cela indique que la JVM n'arrive pas à allouer de la mémoire légitime

- Xmx est trop grand :
 - Diminuer ou acheter de la RAM
- A priori, n'arrive pas si on a fixé,
Xms=Xmx



Algorithmes de collecte

Pour un serveur, on utilise l'algorithme de collecte minimisant les temps de pause.

- Si $< \text{JD7u4}$,
 -XX:+UseConcMarkSweepGC
- Si $> \text{JDK7u4}$
 -XX:+UseConcMarkSweepGC
 Ou
 -XX:+UseG1GC



Options pour évaluer la performance des collectes

Options des commandes en ligne :

- XX:+PrintGCDetails** : affiche pour chaque collecte la taille des objets vivants avant et après la collecte, l'espace libre pour chaque génération et la durée de la collecte
- XX:+PrintGCTimeStamps,**
-XX:+PrintGCDateStamps : Affiche le timestamp du démarrage de la collecte ou la date
- Xloggc:"<path to log>"** : Redirige les traces vers un fichier
- verbose:gc** : Encore plus d'infos



Exemple

```
4.316: [GC [PSYoungGen: 84092K->963K(84800K)] 110155K->27234K(127808K),  
0.0010980 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
```

- **4.316** : Le timestamp par rapport au démarrage de l'application
- **GC** : Type de la collection (GC ou Full GC)
- **PSYoungGen** : est le nom du collecteur
- **84092K** : Espace occupé par la jeune génération avant la collecte
- **963k** : Espace occupé par la jeune génération après la collecte
- **84800k** : Espace disponible pour la jeune génération après la collecte
- **110155k** : Espace occupé total avant la collecte
- **27234k** : Espace occupé total après la collecte
- **127808k** : Total espace disponible de la Heap
- **0.0010980 secs** : Temps pris par la collecte

=> La jeune génération a été vidée intégralement, très peu d'objets ont été copiés dans la vieille génération.

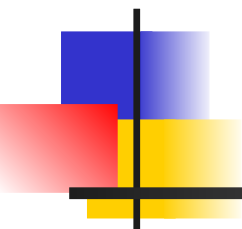


Autres options de traces

La rotation des fichiers de logs du GC facilite l'analyse et garantit l'espace disque :

*-XX:+UseGCLogFileRotation -
XX:NumberOfGCLogFiles=10 -
XX:GCLogFileSize=100M*

Les fichiers de logs peuvent être analysés avec l'outil **GCViewer**



Commandes en ligne Java



Résumé des outils standards

La distribution Java fournit donc un ensemble d'outils adaptés pour différents types de problème et de situation :

- les commandes en ligne
- les outils graphiques (*jconsole*, *jvisualvm* et *jmc*)



Commandes en ligne

jps : Affiche les processus JVM (locaux ou distants) pouvant être instrumenté

jmap : Visualisation de la mémoire, Génération d'un dump

jhat : Analyse de la mémoire d'un dump

jcmd : Commande très complète, affichant des informations sur la VM, les threads, les classes.

jcmd <pid> help. Permet de voir toutes les commandes disponibles

jinfo : Affiche la configuration d'une JVM en cours d'exécution. Permet de modifier certains flags dynamiquement

jstack : Affiche les stacktrace des Threads d'une JVM en cours d'exécution (local ou distante) ou à partir d'un dump

jstat : Outil de monitoring et de génération de statistiques (peut disparaître dans les prochaines releases)



Typologie des outils

Ces outils peuvent s'adresser à différents types d'information :

Information basique sur la VM

- Sur les threads
- Sur les classes
- Analyse de la collecte
- Traitement à posteriori d'un heap dump




jmap

L'outil **jmap** est une ligne de commande disponible sous Linux

Il permet d'afficher des statistiques sur la mémoire à partir d'une JVM en cours d'exécution ou d'un fichier dump

- L'option **-heap** permet d'afficher le nom du garbage collector (et donc l'algorithme utilisé), la configuration de la heap et son usage
- L'option **-histo** permet d'afficher pour chaque classe le nombre d'instances en mémoire et l'espace occupé (utile pour repérer les fuites mémoires)
- L'option **-permstat** peut être utilisée pour analyser le pool permanent
- L'option **-finalizerinfo** affiche les objets en attente de finalisation



jmap -heap

Les informations indiquées *jmap -heap* sont :

La configuration de la heap :

MinHeapFreeRatio et ***MaxHeapFreeRatio*** : les pourcentages d'espace libre provoquant l'expansion ou la contraction de la heap

MaxHeapSize : Taille maximale de la heap

NewSize, ***MaxNewSize*** : Taille initiale et maximale de la jeune génération

OldSize : Taille initiale de la vieille génération

NewRatio : Ratio entre la jeune et vieille génération

SurvivorRatio : Ratio entre un pool survivor et le pool Eden

PermSize, ***MaxPermSize*** : Taille initiale et maximale de la zone permanente

L'usage instantanée de la heap :

Pour chaque pool, la capacité, l'utilisation et le pourcentage utilisé



Statistiques par classe

La commande **jmap** permet d'obtenir un histogramme de l'utilisation de la heap par classe incluant le nombre total d'instance et l'espace en octets occupé par chaque classe

```
jmap -histo:live <pid>
```

num	#instances	#bytes	class name

1:	100000	41600000	[LMemLeak\$LeakingClass;
2:	100000	2400000	MemLeak\$LeakingClass
3:	12726	1337184	<constMethodKlass>
4:	12726	1021872	<methodKlass>
5:	694	915336	[Ljava.lang.Object;
6:	19443	781536	<symbolKlass>



Informations sur la VM

Les outils Java permettent d'obtenir plusieurs informations basiques :

La durée d'un process Java

```
jcmd <pid> VM.uptime
```

Les propriétés de la JVM

```
jcmd <pid> VM.system_properties
```

```
jinfo -sysprops <pid>
```

La version de la JVM

```
jcmd <pid> VM.version
```

La commande en ligne exécutée

```
jcmd <pid> VM.command_line
```

Les flags effectifs

```
jcmd <pid> VM.flags [-all]
```




Autres informations

Threads

jstack <pid>

jcmd <pid> Thread.print

Classes

jstat -class <pid>

Garbage collector

jstat -gcutil <pid>

Dump

jhat

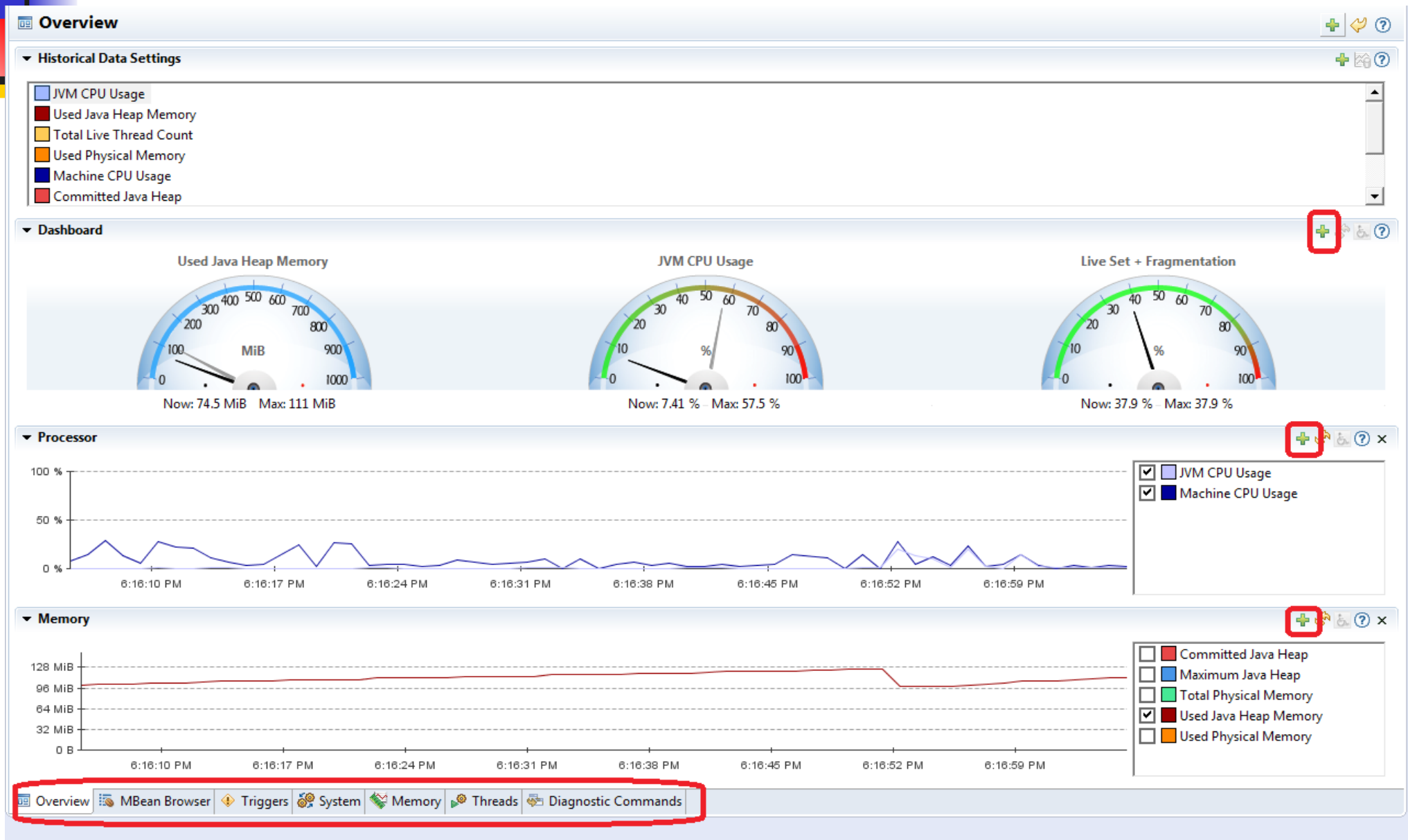


Outil graphique *jmc*

Java Mission Control (jmc) est constitué de :

- Une console JMX pour faire du monitoring en temps-réel
- Java Flight Recorder qui collecte des données sur la JVM et l'application pour une analyse à posteriori
- Des plug-ins optionnels

Console JMX





Particularités JavaEE



Introduction

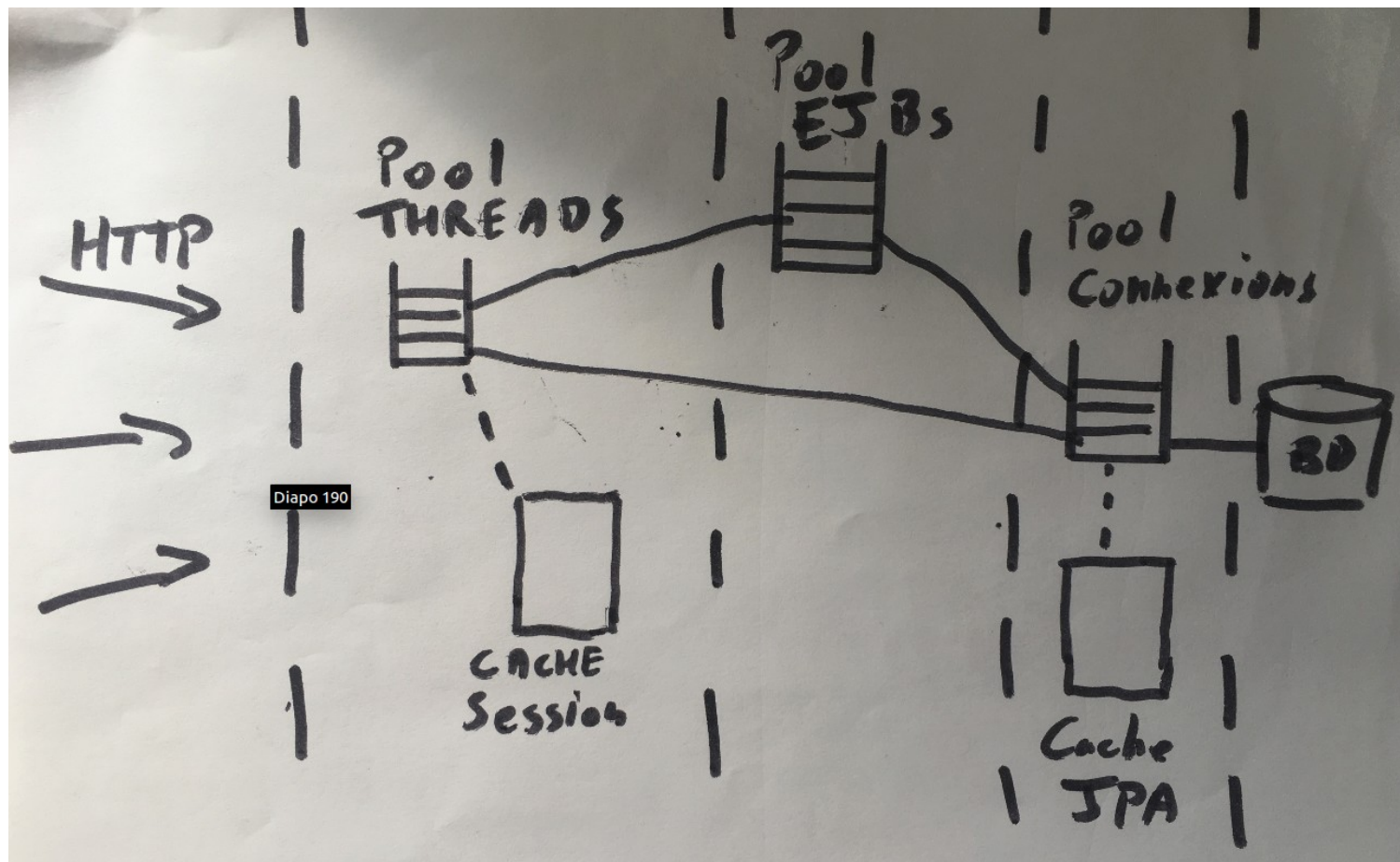
Une application Java EE est une application Java profitant de services techniques offertes par le serveur

L'application interagit avec des objets administrés et configurés sur le serveur : Exemple : pool de connexions BD

Ces objets sont principalement de 2 types :

- Des **pools** : Pattern permettant la réutilisation qui limite les allocations et l'empreinte mémoire
- Des **caches** : Pattern accélérant les temps de réponse

Pools et Cache





Minimiser le volume de sortie

Du point de vue du conteneur web, la performance s'améliore si les réponses HTTP sont moins volumineuses :

- => Éviter les espaces et les tabulations dans la réponse
- => Il est plus efficace de servir les CSS et le javascript avec un seul gros fichiers plutôt que plusieurs petits fichiers
- => Compresser la sortie en utilisant le mime-type zip ou gzip
- => Ne pas utiliser la compilation JSP dynamique



Session HTTP

Les données en session restent par défaut 30 minutes en mémoire, cela peut avoir des impacts sur la mémoire et sur le GC

- => Minimiser les données stockées en session
- => Sérialiser sur le disque des sessions inactives
- => Minimiser le délai de 30 minutes
- => Si configuration HA, attention au mode de réplication utilisé par le serveur (réplication totale ou incrémentale)



Pools de Threads

Les serveurs applicatifs utilisent plusieurs pools de threads : un pour les requêtes servlet, un pour les requêtes EJBs, un autre pour les requêtes JMS.

- Certains serveurs permettent même que plusieurs pools puissent être utilisés par type de trafic

Les pools séparés permettent une priorisation des requêtes lorsque tous les CPU sont utilisés.

=> le pool avec le plus de threads a plus de chances d'obtenir le CPU

=> Il faut analyser le type de trafic de son serveur applicatif pour trouver les bonnes proportions entre les différents pools



EJBs stateful

Les EJBs stateful maintiennent un état de leur conversation avec le client. Ils sont maintenus dans un cache mémoire. Si besoin est, le serveur applicatif peut « passer » un EJB stateful afin de limiter la taille du cache

En pratique, la passivation est rarement souhaitable en terme de fonctionnalité et a un impact sur les performances

=> Les caches doivent être configurés afin d'éviter la passivation



Local ou Remote

Il est toujours plus avantageux d'utiliser l'interface locale de l'EJB plutôt que l'interface distante

L'accès distant nécessite la sérialisation/désérialisation des arguments

Les EJBs distants supportent IIOP mais généralement les serveurs applicatifs proposent d'autres protocoles propriétaires plus rapides



Sérialisation

Dans un contexte Java EE, la sérialisation est utilisée intensivement en particulier pour :

- Sauvegarder l'état d'une session HTTP
- Passer les arguments à un EJB distant

Le JDK propose un mécanisme par défaut pour sérialiser les objets qui implémente `Serializable` ou `Externalizable`

Ce mécanisme peut être optimisé dans la plupart des cas ... mais cela nécessite du travail et peut introduire des bugs



Pools de connexions

Les connexions vers la BD sont en général réutilisées grâce à un pool de connexions.

Bien dimensionner ce pool a naturellement beaucoup d'impact sur les performances

En général, les valeurs du pool correspondent également à celui des threads traitant les requêtes



XA-Transaction

Les transactions XA permettent d'étendre la transaction sur différentes ressources.

Elles sont coûteuses en terme de performance

La plupart des serveurs JavaEE optimise l'algorithme XA afin que la dernière ressources à commiter utilise un protocole de validation simple (Exemple : *Last Resource Commit Optimization (LRCO)*)

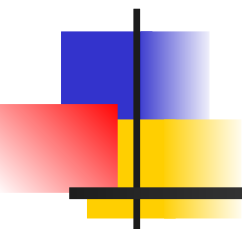
Si le serveur ne propose pas ce genre d'optimisation et que l'on utilise une simple base de données, éviter les transactions XA



Caches pour la persistance

2 types de cache sont utilisés pour la couche de persistance :

- Le **prepared statement cache** cache le résultat de la compilation d'une requête. Ainsi si une même String SQL est exécutée, elle profite du cache. Ce cache est dimensionné par les administrateurs pour un pool de connexions.
- Le **cache JPA** cache le résultat d'une requête. Si la même requête arrive, la base n'est pas sollicitée. L'utilisation du cache est définie au niveau applicatif et le cache implémenté par le serveur peut être configurée par l'administrateur



Pools et caches dans JBoss



Pools

JBoss comme tous les autres serveurs applicatifs introduit des pools à plusieurs niveaux :

- Pool de connexions BD
- Pool des différents EJBs
- Pool de threads

Les pools de threads sont utilisées par plusieurs sous-système de Jboss. Des configuration par défaut existe mais elles peuvent être personnalisées via le sous-système ***thread***



Usage

Le sous-système de thread permet de configurer des pools en en utilisant 6 types prédéfinis :

- ***unbounded-queue-thread-pool*** : Pool de threads illimité, acceptant toujours des tâches
- ***bounded-queue-thread-pool*** : Pool de threads pouvant rejeter des tâches
- ***blocking-bounded-queue-thread-pool*** : Pool pouvant bloquer les tâches
- ***queueless-thread-pool*** : Démarre tout de suite la tâche en asynchrone
- ***blocking-queueless-thread-pool*** : Démarre tout de suite ou bloque la tâche
- ***scheduled-thread-pool*** : Exécution de tâche programmé

Une fois défini, le pool peut être utilisé dans un autre sous-système de JBoss



Infinispan

Les techniques de cache permettent d'améliorer les performances des accès aux données

JBoss utilise le sous-système Infinispan pour toutes les fonctionnalités de cache.

Infinispan est très sollicité en mode cluster

En mode simple, il est utilisé pour la session web, le cache de second niveau JPA



Monitoring JBoss

La console d'administration
Gestion des traces
L'outil CLI



Comparaison CLI/Console

CLI :

- Incontournable pour les administrateurs expérimentés
- Permet d'atteindre tous les attributs du serveur (métriques par exemple)
- Les opérations peuvent être automatisées sous forme de macro ou de batch

Console :

- Tâche d'administration basique facilitée.
- Gestion des ressources de haut-niveau d'un domaine



Interface web



Console d'administration

L'interface web est une application GWT qui utilise l'API HTTP de gestion.

L'API HTTP offre deux contextes :

- Un pour exécuter les opérations d'administration :

http://<host>:9990/management

- Un autre pour accéder à la console :

http://<host>:9990/console



Onglets standalone

Dans le mode standalone, la console propose trois onglets principaux :

- L'onglet **Configuration** est utilisée pour la configuration du serveur
- L'onglet **Deployments** permet de gérer les déploiements
- L'onglet **Runtime** est dédié au monitoring

[Home](#)[Deployments](#)[Configuration](#)[Runtime](#)[Administration](#)

Subsystems <<

DATASOURCES

XA DATASOURCES

▼ Connector

JCA

Datasources

Resource Adapters

Mail

> Container

> Core

> Infinispan

> Security

> Web

Diapo 190

General Configuration

Interfaces

Socket Binding

Paths

System Properties

JDBC Datasources

JDBC datasource configurations.

Available Datasources

[Add](#)[Remove](#)[Disable](#)

Name	JNDI	Enabled?
ExampleDS	java:jboss/datasources/ExampleDS	✓

<< < 1-1 of 1 > >>

[Attributes](#)[Connection](#)[Pool](#)[Security](#)[Properties](#)[Validation](#)[Timeouts](#)[Need Help?](#)[✎ Edit](#)**Name:** ExampleDS**JNDI:** java:jboss/datasources/ExampleDS**Is enabled?:** true**Statistics enabled?:** false**Datasource Class:****Driver:** h2**Driver Class:**



Onglets domaine

Le mode domaine apporte les changements suivants :

- L'onglet **Configuration** permet de gérer plusieurs profils
- L'onglet **Domain** permet de gérer les serveurs et leurs affectations aux groupes de serveur

Topologie du domaine

RED HAT JBOSS® ENTERPRISE APPLICATION PLATFORM 6.4.0.GA

Messages: 0 Red Hat Access Search dthibau

Home Deployments Configuration **Domain** Runtime Administration

Domain << TOPOLOGY EXTENSIONS Refresh

Overview

Server Groups

Host Configuration

Host:

master

Server Configurations Diapo 190

JVM Configurations

Interfaces

Host Properties

Hosts, groups and server instances

An overview of all hosts, groups and server instances in the domain.

Hosts → Groups ↓		
master	Domain: Controller	★
main-server-group Profile: full	server-one Socket Binding: full-sockets Ports: +0	✓
	server-two Socket Binding: full-sockets Ports: +150	✓
other-server-group Profile: full-ha	server-three	⊘

<< < 1-1 of 1 > >>

2.5.5.Final-redhat-1 Tools Settings



Onglet Runtime

L'onglet *Runtime* permet d'accéder aux métriques les plus importants pour surveiller le serveur :

- Infos JVM
- Métriques sur les datasources
- Métriques JPA si préalablement autorisés dans *persistence.xml*
- Métriques JMS
- Métriques transactions et logs des transactions
- Métriques web et services web
- Consulter les logs, les propriétés d'environnement, l'arbre JNDI

Certains métriques ne s'affichent que si le monitoring a été activé au niveau de la configuration

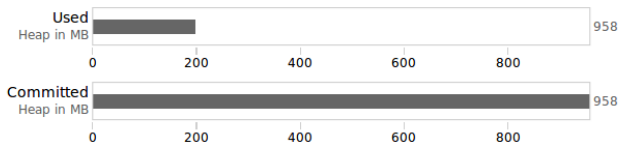
Infos JVM

Principalement, état de la heap et le nombre de threads courant

Operating System: Linux 4.4.0-92-generic
Processors: 4
JVM Uptime: 1 min, 24 s

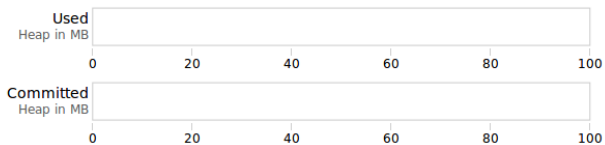
Heap Usage

Max 958
Used 199
Committed 958



Non Heap Usage

Max 0
Used 94
Committed 103



Thread Usage

Live 106
Daemon 26





Visualiseur de log

Fonctionnalité de filtres + téléchargement

Log Viewer

Log files of selected server

Filter:

[Download](#)[View](#)

▲ Log File Name	Date - Time (UTC)	Size (MB)
server.log	2017-09-08T10:12:49.000+0200	0.02
server.log.2017-09-07	2017-09-07T19:23:31.000+0200	0.06

« « 1-2 of 2 » »

DataSource

Principalement, état du pool de connexions et statement cache

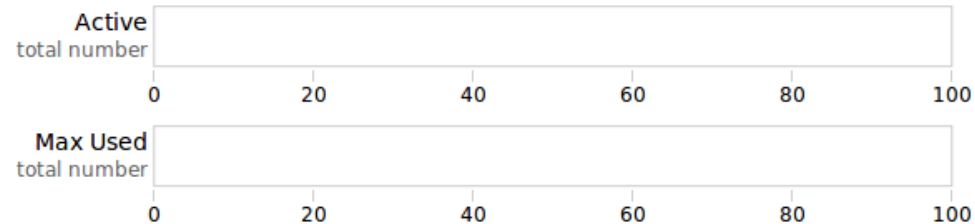
Connection Pool

[Refresh Results](#)

Available
Connections 0

Active 0

Max Used 0

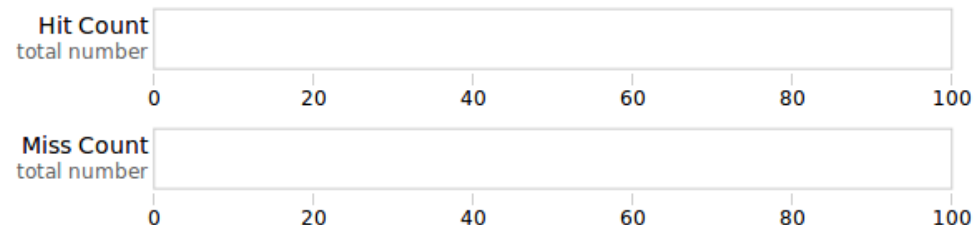


Prepared Statement Cache

Access Count 0

Hit Count 0

Miss Count 0





JPA

Informations assez précise sur l'usage d'une unité de persistance :

- Nombre de sessions ouvertes/fermées
- Nombre de transactions
- Nombre de requêtes, temps max d'exécution
- Caches Hibernate (2nd Level et Query Cache)
-



JMS

Il est possible de voir les métriques
relatifs aux différentes destinations
(File et Topic) :

- Nombre de consommateur
- Nombre d'abonnés
- Nombre de messages
- ...



Transactions

Le nombre de transactions :

- Validées
- Annulées
- En time-out

Typologie des échecs :

- Application (Exception applicative durant la transaction)
- Ressource

Les logs de recovery sont également disponibles



Web et services Web

Pour le web, uniquement le nombre de requêtes par connecteurs et le nombre de requêtes en erreur

Pour chaque endpoint d'un web service, le nombre de réponses et le nombre de fautes WSDL générées



Gestion des traces



Terminologie

- ♦ **Logger ou log category**: Un objet *logger* produit des messages. Il est associé à un package Java. Ils sont donc organisés hiérarchiquement (Tout le monde hérite de *rootLogger*).
- ♦ **Handler** : Chaque *logger* peut être associé à plusieurs *handler*. Un *handler* permet d'écrire le message sur une sortie. Par exemple, le *FileHandler* est associé à fichier de log.
- ♦ **Formatter** : Un handler peut optionnellement utiliser un formatter pour localiser et formater le message avant de le publier sur son flux I/O.

Des niveaux de trace peuvent être affectés aux *loggers* et aux *handlers* afin de filtrer les messages



Sous-système de logging

La configuration du système de trace est effectué via le sous-système de logging

```
<extension module="org.jboss.as.logging"/>
```

```
<subsystem xmlns="urn:jboss:domain:logging:1.0">
```

Elle est composée de plusieurs parties :

- La configuration des *handlers* . Chaque *handler* déclare le format de trace et de sortie
- Les déclarations des *logger* et du *logger* racine (*log categories*).
- Chaque *logger* référence un ou plusieurs *handler*



Example

```
<subsystem xmlns="urn:jboss:domain:logging:1.0">
  <console-handler name="CONSOLE" autoflush="true">
    <level name="DEBUG"/>
    <formatter><pattern-formatter pattern="%d{HH:mm:ss,SSS} %-5p [%c] (%t) %s%E%n"/></formatter>
  </console-handler>
  <periodic-rotating-file-handler name="FILE" autoflush="true">
    <level name="INFO"/>
    <formatter><pattern-formatter pattern="%d{HH:mm:ss,SSS} %-5p [%c] (%t) %s%E%n"/></formatter>
    <file relative-to="jboss.server.log.dir" path="server.log"/>
    <suffix value=".yyyy-MM-dd"/>
  </periodic-rotating-file-handler>
  <logger category="com.arjuna"><level name="WARN"/></logger>
  [...]
  <root-logger>
    <level name="DEBUG"/>
    <handlers>
      <handler name="CONSOLE"/>
      <handler name="FILE"/>
    </handlers>
  </root-logger>
</subsystem>
```



Emplacements par défaut mode domaine

Mode standalone :

./standalone/log/boot.log

./standalone/log/server.log

Mode domaine :

Contrôleur d'hôte :

./domain/log/host-controller/boot.log

Contrôleur de processus

./domain/log/process-controller/boot.log

Instance de serveur

./domain/servers/<instance-name>/log/boot.log

./domain/servers/<instance-name>/log/server.log



Handler directement disponibles

console-handler : Écriture sur la console

periodic-rotating-file-handler : Écriture dans un fichier tournant après un certain temps (la journée par défaut)

size-rotating-file-handler : Écriture dans un fichier tournant en fonction de la taille du fichier

asynchronous : *Handler* composite s'attachant à d'autres *handlers* permettant une écriture en asynchrone



Configuration des *Logger*

Un objet *Logger* est responsable de tracer les messages d'un composant particulier de l'application.

Les *loggers* sont nommés avec le nom d'un package ou d'une classe

Ils sont donc hiérarchiques et héritent tous du *logger* racine.

Les niveaux de trace peuvent être surchargés par les descendants



TP : Logs

❖ Logs :

- Comprendre les niveaux de logs, les sorties
- Modifier le niveau de différents handler et logger



L'outil CLI



Utilisation du CLI

Lancement du CLI en mode interactif

<JBASS_HOME>/bin/jboss-cli.sh(.bat)

Options de lancement :

- **--connect** : pour se connecter immédiatement à un serveur
- **--controller**=host:port : à utiliser avec l'option '--connect', permet d'indiquer l'adresse et le numéro de port (par défaut localhost et 9999)
ex : ./jboss-cli.sh --connect --controller=192.168.0.28:9999
- **--file** : exécution des commandes se trouvant dans le fichier indiqué
- **--command** : exécution de la commande indiquée
- **--commands** : exécution des commandes passées en paramètre sous la forme d'une chaîne avec des virgules comme séparateurs
- **--user** et **--password** : pour éviter le prompt
- **--gui** : pour utiliser le mode graphique



Le mode interactif

Utiliser la tabulation

Utiliser les flèches

Aide détaillée d'une commande

```
[domain@192.168.0.28:9999 /] deploy --help
```

SYNOPSIS

```
    deploy (file_path [--script=script_name] [--name=deployment_name] [--runtime-  
name=deployment_runtime_name]  
           [--force | --disabled] [--unmanaged])  
    | --name=deployment_name  
    | --server-groups=group_name (,group_name)* | --all-server-groups  
    | --headers={operation_header (;operation_header)*}]
```

DESCRIPTION

Deploys the application designated by the file_path or enables an already existing


...



Principes de bases du CLI (1)

La logique d'utilisation du CLI est la suivante :

- chaque entrée dans les fichiers de configuration est identifiée par son chemin
- sur chaque entrée des 'opérations' peuvent être exécutées
- en mode interactif, il est possible de naviguer dans la configuration du serveur :
 - reprend les commandes classiques de navigation dans les répertoires : ls , cd et pwd
 - le point de départ '/', est la racine de la configuration



pwd, cd, ls

```
[standalone@localhost:9999 /] pwd
/
[standalone@localhost:9999 /] ls
core-service          deployment            extension
interface             path                 socket-binding-group
subsystem             system-property      launch-type=STANDALONE
management-major-version=1
management-minor-version=2
namespaces=[]         process-type=Server  name=blc4-ubuntu
product-version=6.0.0.GA
profile-name=undefined
release-version=7.1.2.Final-redhat-1
running-mode=NORMAL  product-name=EAP
server-state=running
release-codename=Steropes
schema-locations=[]

[standalone@localhost:9999 /] cd deployment
[standalone@localhost:9999 deployment] ls
postgresql-9.1-902.jdbc4.jar  tp42.ear              tpDatasource.war
[standalone@localhost:9999 deployment]
```





Principes de base (2)

Chaque élément de la configuration peut avoir des attributs et des sous-éléments (structure classique du format XML)

- la commande 'ls' affiche les deux
- La commande 'ls -l' affiche plus de détail

Sur chaque élément des '**opérations**' peuvent être exécutées :

- la commande ***read-operation*** affiche la liste des opérations de l'élément courant



ls, read-operation

```
[standalone@localhost:9999 deployment] ls
postgresql-9.1-902.jdbc4.jar    tp42.ear                    tpDatasource.war
[standalone@localhost:9999 deployment] cd tp42.ear
[standalone@localhost:9999 deployment=tp42.ear] ls

subdeployment
subsystem
content=[{"hash" => bytes { 0x53, 0x62, 0x92, 0x23, 0xdb, 0x21, 0xbe, 0x79, 0x8a, 0x37, 0x91,
0x0b, 0xe1, 0x53, 0x86, 0x14, 0x72, 0x60, 0x5e, 0x6a }}}]
enabled=false
name=tp42.ear
persistent=true
runtime-name=tp42.ear
status=STOPPED
[standalone@localhost:9999 deployment=tp42.ear]
```

```
[standalone@localhost:9999 deployment=tp42.ear] read-operation

add                deploy                read-attribute
read-children-names    read-children-resources    read-children-types
read-operation-description    read-operation-names    read-resource
read-resource-description    redeploy                remove
undefine-attribute        undeploy                whoami
write-attribute
```



Syntaxe des opérations

La syntaxe d'exécution des opérations est :

<nom_operation>(nom_paramètre=valeur_paramètre,p2=v2)

Exemples :

:undeploy

:read-attribute(name="status")

:write-attribute(name="max-pool-size",

:write-attribute(name="max-pool-size", value="20")

Documentation d'une opération :

:read-operation-description(name="write-attribute")




Format de la réponse

La réponse est au format JSON. Elle indique toujours un **outcome** et éventuellement les paramètres de retours de l'opération

Exemple :

```
:whoami
{
  "outcome" => "success",
  "result" => {"identity" => {
    "username" => "$local",
    "realm" => "ManagementRealm"
  }}
}
```



Invocation d'opérations

En mode interactif, sans autres précisions, les opérations s'exécutent sur l'élément courant

Possibilité d'exécuter une commande sur un élément précis :

- nécessaire en mode non interactif, les commandes de navigation n'étant pas disponibles
- syntaxe : **<id_élément>:<nom_opération>**
- Exemple :

```
/subsystem=datasources/data-source=TestMySQL:write-attribute(name="max-pool-size", value="20")
```



Distinction commande et opération

La documentation JBoss distingue bien les *commandes* des *opérations*.

- Une **opération** est contextuelle à l'emplacement où l'on se trouve. Elle commence par :
- Une **commande** peut se lancer de n'importe où, elle consiste généralement en une opération transverse



Commandes haut-niveau

cn (ou cd) : Modification de l'emplacement courant dans l'arbre des ressources

deploy : Déploiement d'applications

help (ou h) : Aide

history : Accès à l'historique des commandes

ls : Lister le contenu du nœud courant

pwn (ou pwd) : Affiche le nœud courant;

quit (ou q) : Quitter CLI;

undeploy : Repli d'une application;

version : Affichage de la version et des informations d'environnement

add-jms-queue/remove-jms-queue : Création/Suppression d'une file JMS

add-jms-topic/remove-jms-topic : Création/Suppression d'un topic

add-jms-cf/remove-jms-cf : Création/suppression d'une usine à connexion JMS

data-source : Ajout/Suppression/Modification d'une source de données non XA

xa-data-source : Ajout/Suppression/Modification d'une source de données XA



Arbre des ressources

Les ressources administrables sont organisées sous forme **d'arbre**. (XML)

Le chemin dans l'arbre correspondant à une ressource est appelée l'**adresse**

Elle est composée d'une liste ordonnée de paires clé/valeur partant de la racine de l'arbre jusqu'à la ressource

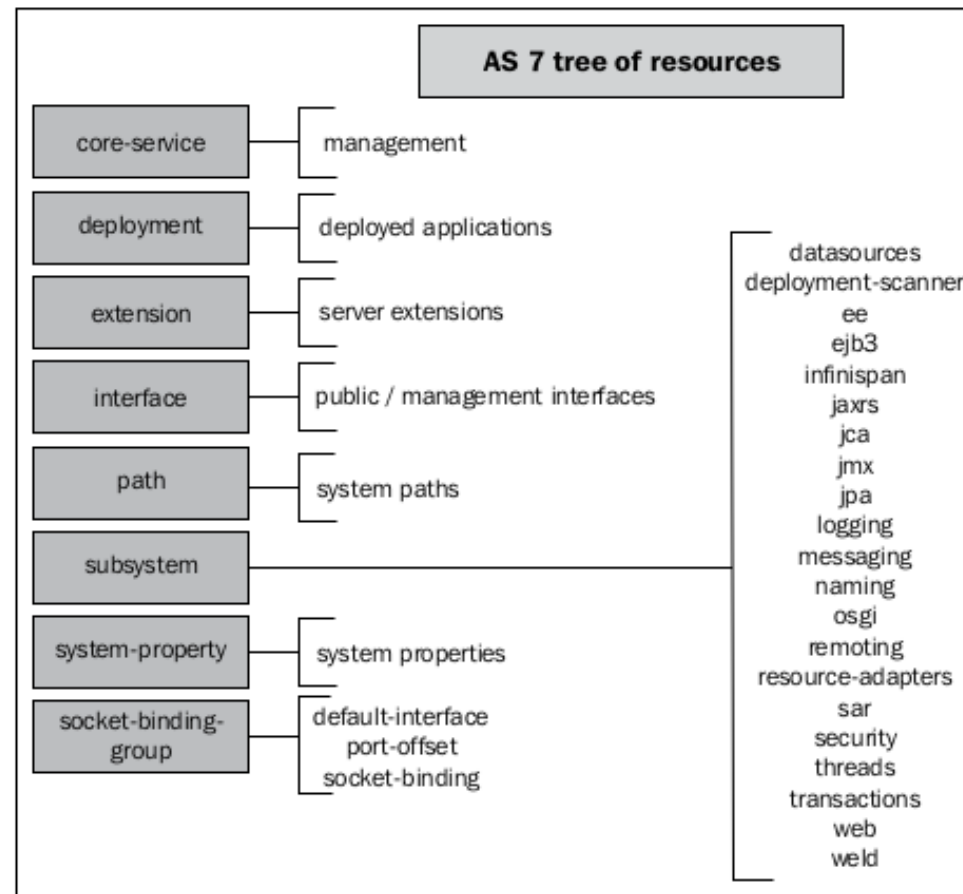
- La clé correspond au type de la ressource dans le contexte de son parent.
- La valeur est le nom d'une ressource particulière du type correspondant.

Les éléments sont en général séparés par un '/' et le '=' sépare la clé de la valeur.
(Lorsque l'on utilise l'API HTTP, '=' est remplacé par '/')

Par exemple :

- /subsystem=messaging/jms-queue=testQueue
- http://localhost:9990/management/subsystem/messaging/jms-queue/
testQueue

Arbre des ressources en standalone





Arbre des ressources en mode domaine

extension : extensions disponibles dans le domaine

path : chemins disponibles dans le domaine

system-property : propriétés systèmes du domaine

profile : Configuration de sous-système applicable à un groupe de serveurs

subsystem : Configuration des sous-système appartenant à un profil

interface : Configurations des interface

socket-binding-group : configuration des socket bindings pouvant être appliqué au groupe de serveur

socket-binding : configuration individuelle des sockets

deployment : déploiement disponible applicable à des groupes de serveurs

server-group : Les configurations des groupes de serveur

host : Les contrôleurs d'hôtes. Chaque enfant représente la ressource racine d'un hôte particulier. Les informations sont stockées dans le *host.xml* correspondant.



Données d'une ressource

Une ressource est constituée de différents types d'information :

- **description** : Description de la ressource
- **attributes** : Les attributs configurables de la ressource
- **operations** : Les opérations d'administration disponibles sur la ressource
- **children** : Les ressources filles
- **head-comment-allowed** : Booléen indiquant si les commentaires XML sont permis avant l'élément XML de la ressource. (true par défaut)
- **tail-comment-allowed** : Booléen indiquant si les commentaires sont permis juste avant la fermeture de la balise correspondant à la ressource



Attributs

Les ressources administrables exposent des informations concernant leur statut sous forme **d'attributs**.

- Les attributs ont un nom et une valeur
- Les attributs sont soit *read-only* ou *read-write*. Les opérations permettant de consulter ou modifier un attribut sont ***read-attribute*** et ***write-attribute***.

Les attributs ont également un ou deux modes de stockage :

- **CONFIGURATION** : L'attribut est stocké dans le fichier XML de configuration.
- **RUNTIME** : L'attribut n'est disponible seulement lorsque le serveur est en cours d'exécution et n'est pas stocké sur le support persistant (Exemple un métrique).



Arborescence configuration et runtime

En mode domaine, on peut distinguer 2 arborescences distinctes :

- Une arborescence de **configuration** qui commence à
`/profile=<nom_profil>`
- Une arborescence de **runtime** qui commence à
`/host=<hote>/server=<serveur>`



Opération modification config

Chaque ressource de configuration à l'exception de la ressource racine a une opération ***add*** et doit avoir une opération ***remove***

Ces opérations sont utilisées pour les mises à jour de la configuration d'un profil



Exemples

Ajout d'un driver

```
/subsystem=datasources/jdbc-driver=postgresql: add(driver-  
name=postgresql,driver-module-name=org.postgresql.jdbc,driver-xa-  
datasource-class-name=org.postgresql.xa.PGXADatasource)
```

Suppression du driver

```
/subsystem=datasources/jdbc-driver=postgresql: remove
```

Ajout d'un log handler

```
/subsystem=logging/periodic-rotating-file-  
handler=FILE_QS_TRACE: add(suffix=".yyyy.MM.dd",  
file={"path"=>"quickstart.trace.log", "relative-  
to"=>"jboss.server.log.dir"})
```

Ajout d'un log category

```
/subsystem=logging/logger=com.your.package.name: add(level=DEBUG)
```



Mode batch

Par défaut les commandes sont exécutées immédiatement

Possibilité d'exécuter une série de commandes en mode batch :

- si une commande échoue toutes les précédentes sont annulées
- la commande 'batch' permet d'entrer en mode batch
 - les commandes saisies ne sont plus exécutées immédiatement
 - la commande 'run-batch' lance l'exécution des commandes historisées



Example

```
[domain@192.168.0.28:9999 /] batch
[domain@192.168.0.28:9999 / #] /profile=default/subsystem=datasources/jdbc-driver=mysql:add(driver-
name=mysql,driver-module-name=com.mysql,driver-xa-datasource-class-
name=com.mysql.jdbc.jdbc2.optional.MysqlXADataSource)
#1 /profile=default/subsystem=datasources/jdbc-driver=mysql:add(driver-name=mysql,driver-module-
name=com.mysql,driver-xa-datasource-class-name=com.mysql.jdbc.jdbc2.optional.MysqlXADataSource)
[domain@192.168.0.28:9999 / #] data-source add --profile=default --name=DSMySQL --jndi-
name=java:/ds/Test2MySQL --driver-name=mysql --connection-url=jdbc:mysql://192.168.0.49/test?
user=root&password=admin
#2 data-source add --profile=default --name=DSMySQL --jndi-name=java:/ds/Test2MySQL --driver-name=mysql
--connection-url=jdbc:mysql://192.168.0.49/test?user=root&password=admin
[domain@192.168.0.28:9999 / #] data-source enable --profile=default --name=DSMySQL
#3 data-source enable --profile=default --name=DSMySQL
[domain@192.168.0.28:9999 / #] run-batch
The batch executed successfully.
```



:read-resource

L'opération ***:read-resource*** permet de lire les attributs de la ressource courante.

Elle est très utile pour le monitoring

Elle prend, entre autre, les options suivantes :

- ***recursive, recursive-depth*** : Permet de retourner les attributs des ressources filles
- ***include-runtime*** : si les métriques de runtime doivent être retournés

Exemple :

```
// Récupérer toutes les informations de configuration et de runtime  
:read-resource(include-runtime=true, recursive=true, recursive-depth=10)
```



Commandes de monitoring

Statut du serveur :


```
[standalone@localhost:9990 /] :read-attribute(name=server-state)
{ "outcome" => "success",
  "result" => "running" }
```

Statut d'un déploiement

```
[standalone@localhost:9990 /] /deployment=example.ear :read-attribute(name=status)
{ outcome" => "success",
  "result" => "OK" }
```

Métriques JVM

```
[standalone@localhost:9990 /]/core-service=platform-mbean/type=memory :read-
attribute(name=heap-memory-usage)
{ "outcome" => "success",
  "result" => {
    "init" => 67108864L,
    "used" => 58717032L,
    "committed" => 91910144L,
    "max" => 518979584L
  } }
```



Commandes de monitoring (2)

Application web :


```
[standalone@localhost:9990 /] /deployment=example.ear/subdeployment=example-web.war/subsystem=undertow :read-attribute(name=active-sessions)
{ "outcome" => "success",
  "result" => 3 }
```

Data sources

```
[standalone@localhost:9990 /]
/subsystem=datasources/data-source=ExampleDS/statistics=pool :read-resource(recursive=true,
include-runtime=true)
{ "outcome" => "success",
  "result" => {
    "ActiveCount" => "1",
    "AvailableCount" => "20",
    ...
  }
}
```

JPA

```
[standalone@localhost:9990 /] /deployment=web-application-example.ear/subsystem=jpa/hibernate-
persistence-unit=web-application-example.ear#primary :read-resource(include-runtime=true,
recursive=true)
{
  "outcome" => "success",
}
```



Commandes de monitoring (3)

EJB :

```
[standalone@localhost:9990 /] /subsystem=ejb3 :write-attribute(name=enable-statistics, value=true)
deployment=example.ear/subdeployment=example-ejb.jar/subsystem=ejb3/stateless-session-
bean=BlogEntryDaoBean :read-resource(include-runtime=true, recursive=true)
{
    "outcome" => "success",
    "result" => {
        ...
    }
}
```

Pool des instances

```
[standalone@localhost:9990 /] /subsystem=ejb3/thread-pool=default :read-resource(include-
runtime=true, recursive=true)
{
    "outcome" => "success",
    "result" => {
        ...
    }
}
```

JPA

```
[standalone@localhost:9990 /] /deployment=web-application-example.ear/subsystem=jpa/hibernate-
persistence-unit=web-application-example.ear#primary :read-resource(include-runtime=true,
recursive=true)
{
    "outcome" => "success",
    "result" => {
        ...
    }
}
```



Tiers Web

Configuration sous-système
Configuration session
Pool de threads



Intégration Tomcat

JBoss intègre un serveur Web adapté aux applications moyennes

C'est une intégration de Jakarta Tomcat basée sur la version 7.0 qui implémente :

- Servlet 3.0
- JSP 2.2
- et inclut une implémentation de Java Server Faces 2.1

Au dessus du container web, peuvent s'exécuter d'autres frameworks comme le moteur de services Web (Apache CXF)

La documentation la plus complète est disponible sous le projet *JBossWeb*



Configuration

La configuration du service Web est constitué de 4 parties :

- les connecteurs
- Les ressources statiques
- la configuration JSP
- les serveurs virtuels

L' extension nécessaire est :

```
<extension module="org.jboss.as.web" />
```




Exemple

```
<subsystem xmlns="urn:jboss:domain:web:1.0"
  default-virtual-server="default-host">

  <connector name="http" scheme="http" protocol="HTTP/1.1"
    socket-binding="http"/>

  <virtual-server name="default-host" enable-welcome-root="true">
    <alias name="localhost" />
    <alias name="example.com" />
  </virtual-server>
</subsystem>
```



Connecteur

La configuration par défaut ne précise que les attributs principaux, laissant les autres attributs aux valeurs par défaut.

Cette configuration peut convenir pour des applications simples



Types de connecteurs

3 types de connecteurs sont disponibles :

- Connecteurs **HTTP** : Connecteur par défaut sur le port 8080
- Connecteur **HTTPS** : SSL
- Connecteurs **AJP** : Utilisé avec *mod_jk*, *mod_proxy* et *mod_cluster*



Attributs

name : Le nom du connecteur

protocol : Soit HTTP/1.1 (défaut) ou AJP/1.3, ou HTTPS

enable-lookups : Si *true*, appels à *getRemoteHost()* pour effectuer des DNS lookups afin de déterminer l'identité du client. *false* augmente les performances

enabled : activation du connecteur

executor : représente le pool de Thread pouvant être partagé entre différents composants du serveur Web

max-connections : Le maximum de threads traitant les requêtes pouvant être créé \Leftrightarrow maximum de requêtes simultanées

max-post-size : Limite en taille des requêtes HTTP POST, Par défaut 2 MB.

max-save-post-size : Limite en taille d'une requête POST stocké par le container pendant l'authentification (4 KB).



Attributs (2)

proxy-name : Serveur proxy

proxy-port : Port du proxy

redirect-port : Port de redirection, si le connecteur reçoit une requête non-SSL correspondant à une contrainte de sécurité SSL

scheme : Le nom du protocole retourné par *request.getScheme()* (par défaut *http*)

secure : la valeur retournée par *request.isSecure()*
(Par défaut *false*)

socket-binding : l'interface pour ce connecteur



Connecteur HTTP APR

JBoss Web utilise le connecteur Coyote HTTP/1.1 basé sur Java. Ce connecteur est très stable et mature.

Cependant, pour des systèmes à forte charge, il peut être intéressant d'utiliser le connecteur **APR** qui repose sur des librairies natives

APR utilise 2 mécanismes pour augmenter les performances :

- Service de fichiers statiques directement à partir de la mémoire système
- Permet d'implémenter des connexions *keep-alive* pour de plus grand nombre de connexions

Dans ce cas, la configuration SSL doit s'effectuer différemment pour utiliser OpenSSL.

7.0.2

Choix du connecteur HTTP

Au démarrage de Tomcat embarqué,
l'AprLifecycleListener essaie de localiser les
bibliothèques natives. Si il ne les trouve pas,
Tomcat se rabat sur le connecteur standard

```
15:50:39,187 INFO [org.apache.catalina.core.AprLifecycleListener] (MSC
service Thread 1-2) The Apache Tomcat Native library, which allows
optimal performance in production environments, was not found on the
java.library.path
```



Installation de APR – 7.0.2

L'installation du connecteur Natif est très simple :

- Télécharger le connecteur JBoss Natif connector du projet JBoss Web (<http://www.jboss.org/jbossweb/downloads/jboss-native-2-0-9>)
- Dézipper la distribution dans le répertoire \$JBOSS_HOME
- Redémarrer JBoss



7.1.1

Avec Jboss 7.1.x, les librairies natives sont incluses dans la distribution. (Voir `$JBOSS_MODULES/org/jboss/as/web/main/lib`). Il suffit alors de les activer dans la configuration

```
<subsystem xmlns="urn:jboss:domain:web:1.1" default-virtual-  
server="default-host" native="true">  
  <connector name="http" protocol="HTTP/1.1" scheme="http"  
    socket-binding="http"/>  
  <virtual-server name="default-host" enable-welcome-root="true">  
    <alias name="localhost"/>  
    <alias name="example.com"/>  
  </virtual-server>  
</subsystem>
```



Vérification

Avant

[org.apache.coyote.http11.**Http11Protocol**] (MSC service thread 1-1) Starting Coyote HTTP/1.1 on http-localhost-127.0.0.1-8080

Après

[org.apache.coyote.http11.**Http11AprProtocol**] (MSC service thread 1-7) Démarrage de Coyote HTTP/1.1 sur http--127.0.0.1-8080



Configuration des connecteurs via CLI

Les connecteurs sont des ressources filles du sous-système *web* et référencent une association de socket particulière.

La création d'un nouveau connecteur nécessite donc la déclaration préalable d'une association de socket

```
[standalone@localhost:9999 /] /socket-binding-group=standard-sockets/socket-binding=custom:add(port=8181)
```

La socket est alors utilisée dans le connecteur

```
[standalone@localhost:9999 /] /subsystem=web/connector=test-connector:add(socket-binding=custom, scheme=http, protocol="HTTP/1.1", enabled=true)
```



Statistiques

bytesSent : Nombre d'octets envoyés par le connecteur

bytesReceived : Nombre d'octet reçu par le connecteur (données POST).

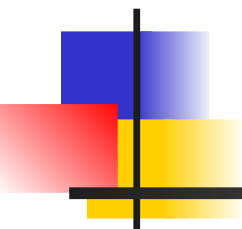
processingTime : Temps de traitement utilisé en ms

errorCount : Nombre d'erreurs lors des requêtes sur le connecteur

maxTime : Temps maximum de traitement d'une requête

requestCount : Nombre de requêtes reçues

```
[standalone@localhost:9999 /] /subsystem=web/connector=http:read-  
  attribute(name=bytesSent, name=requestCount)  
{  
    "outcome" => "success",  
    "result"  => "3"  
}
```



Pool des connecteurs



Pool des connecteurs HTTP

Ajuster le pool de thread associé au connecteur HTTP permet de s'adapter à la charge utilisateur.

Cela est assez difficile de trouver les valeurs optimales mais très important pour les performances.



http-executor

La configuration s'effectue par le thread pool *http-executor*. Les attributs les plus important du connecteur sont ***core-threads*** et ***max-threads***.

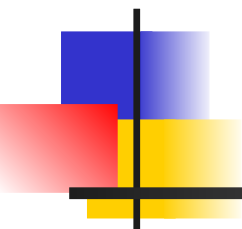
- Si ces valeurs sont trop basses, il se peut qu'il n'y ait pas assez de threads disponibles pour servir les requêtes. Dans ce cas, les requêtes doivent attendre qu'une thread se libère ou partent en timeout.
- D'un autre côté, si l'on augmente le nombre de threads
 - On augmente la consommation mémoire
 - Le système gaspille du temps à switcher entre les threads

La première chose à faire est de s'assurer si certaines requêtes ne sont pas trop longues.



http-executor

```
<subsystem xmlns="urn:jboss:domain:threads:1.0">
  <bounded-queue-thread-pool name="http-executor"
    blocking="true">
    <core-threads count="10" per-cpu="20" />
    <queue-length count="10" per-cpu="20" />
    <max-threads count="10" per-cpu="20" />
    <keepalive-time time="10" unit="seconds" />
  </bounded-queue-thread-pool>
</subsystem>
```

Configuration session



Introduction

Les sessions HTTP peuvent avoir un impact sur les performances et les capacités de charge du serveur.

2 données doivent être prises en compte et optimisées :

- Le volume de données stockée dans la session (Développement)
- Le timeout d'expiration d'une session (Configuration server et application)

De plus, Jboss propose un mécanisme de passivation de session permettant de limiter la mémoire utilisée.



Configuration du time-out

La configuration du timeout peut s'effectuer :

- Au niveau de l'application via le descripteur de déploiement standard web.xml
- Au niveau server (valeur par défaut pour toutes les applications) via le sous-système Web



Exemple

WEB-INF/web.xml

```
<web-app ...>  
<session-config>  
  <session-timeout>20</session-timeout>  
</session-config>  
</web-app>
```

CLI

```
/subsystem=web/:write-attribute(name=default-  
  session-timeout,value=20)
```



Configuration de la passivation

La configuration de la passivation s'effectue dans le descripteur de déploiement spécifique *jboss-web.xml*

L'élément ***max-active-sessions*** permet de limiter le nombre de sessions actives en même temps

L'élément ***passivation-config*** permet de préciser quand une session est passivée. Il précise :

- ***<passivation-min-idle-time>*** : Le minimum de temps d'inaction pour qu'une session soit passivée si le nombre de sessions dépass *max-active-sessions*
- ***<passivation-max-idle-time>*** : La maximum de temps d'inaction pour qu'un session soit systématiquement passivée



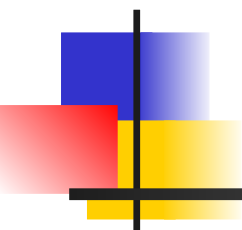
Example

```
<jboss-web version="6.0">  
  <max-active-sessions>20</max-active-sessions>  
  <passivation-config>  
    <use-session-passivation>true</use-session-passivation>  
    <passivation-min-idle-time>60</passivation-min-idle-time>  
    <passivation-max-idle-time>600</passivation-max-idle-  
    time>  
  </passivation-config>  
</jboss-web>
```



Monitoring

```
/deployment=myAppli.war/subsystem=web/  
:read-resource(include-runtime=true)  
{  
  "outcome" => "success",  
  "result" => {  
    "active-sessions" => 0,  
    "context-root" => "/jboss-kitchensink",  
    "duplicated-session-ids" => 0,  
    "expired-sessions" => 0,  
    "max-active-sessions" => 0,  
    "rejected-sessions" => 0,  
    "session-avg-alive-time" => 0,  
    "session-max-alive-time" => 0,  
    "sessions-created" => 0,  
    "virtual-host" => "default-host"  
  }  
}
```



TP



Tiers métier

Tuning des pools d'EJB
Cache EJB stateful



Introduction

La création et la suppression d'EJBs peut être une tâche lourde

Pour optimiser les performances, le conteneur d'EJB crée des pools de beans.

Cela concerne principalement les EJB stateless et MDB.



Configuration

La configuration par défaut est :

```
<bean-instance-pools>
  <!-- stateless -->
  <strict-max-pool name="slsb-strict-max-pool" max-
    pool-size="20" instance-acquisition-timeout="5"
    instance-acquisition-timeout-unit="MINUTES"/>
  <!-- MDB -->
  <strict-max-pool name="mdb-strict-max-pool" max-pool-
    size="20" instance-acquisition-timeout="5" instance-
    acquisition-timeout-unit="MINUTES"/>
</bean-instance-pools>
```



Monitoring

```
:read-resource(include-runtime=true)
{
  "outcome" => "success",
  "result" => {
    "component-class-name" => "MemberRegistration",
    "declared-roles" => [],
    "execution-time" => 0L,
    "invocations" => 0L,
    "methods" => {},
    "peak-concurrent-invocations" => 0L,
    "pool-available-count" => 20,
    "pool-create-count" => 0,
    "pool-current-size" => 0,
    "pool-max-size" => 20,
    "pool-name" => "slsb-strict-max-pool",
    "pool-remove-count" => 0,
    "run-as-role" => undefined,
    "security-domain" => "other",
    "timers" => [],
    "wait-time" => 0L,
    "service" => undefined
  }
}
```



Cache EJB stateful

Les EJB stateful utilisent le service de cache d'InfiniSpan

```
<stateful default-access-timeout="5000"  
  cache-ref="simple"/>
```



Tiers persistance

Configuration des sources de données
Cache EJB stateful



Mécanisme

Jboss 7 permet de configurer les accès aux bases RDBMS en configurant des sources de données JDBC puis en les déployant sur les serveurs ou clusters d'un domaine

Chaque source de données contient un pool de connections BD créées au démarrage du serveur

Les applications effectuent un lookup JNDI global ou dans le contexte applicatif local (*java:comp/env*), afin de demander une connexion. Après utilisation, la connexion est replacée dans le pool.

- **Les informations de connexions aux bases sont généralement externalisées**
- **Une DataSource JDBC permet à l'application de demander une connexion en utilisant un nom logique :**
 - ▶ Les informations de connexions sont renseignées côté serveur d'applications
 - ▶ Remplace l'utilisation de la classe DriverManager de JDBC
 - ▶ Support des pools de connexions de façon portable
 - ▶ Utilisation de JNDI pour retrouver la DataSource :

```
javax.sql.DataSource ds =  
(javax.sql.DataSource)ctx.lookup("jdbc/MaBase");
```


Intérêt des pools de connexions

- **Les connexions et déconnexions aux bases de données sont coûteuses en terme de ressources :**
 - ▶ Temps de connexion
 - ▶ Mémoire utilisée côté client et côté serveur
- **Dans le cas d'une application Web, il serait trop coûteux de réserver une connexion à chaque utilisateur :**
 - ▶ HTTP fonctionne en mode déconnecté (ne permet de savoir quand l'utilisateur quitte l'application)
 - ▶ Le nombre d'utilisateurs peut être important
- **Les serveurs Java EE proposent un mécanisme de pool de connexions qui permet :**
 - ▶ De mettre en commun les connexions
 - ▶ De limiter le nombre de connexions ouvertes

Installation d'un driver JDBC (1)

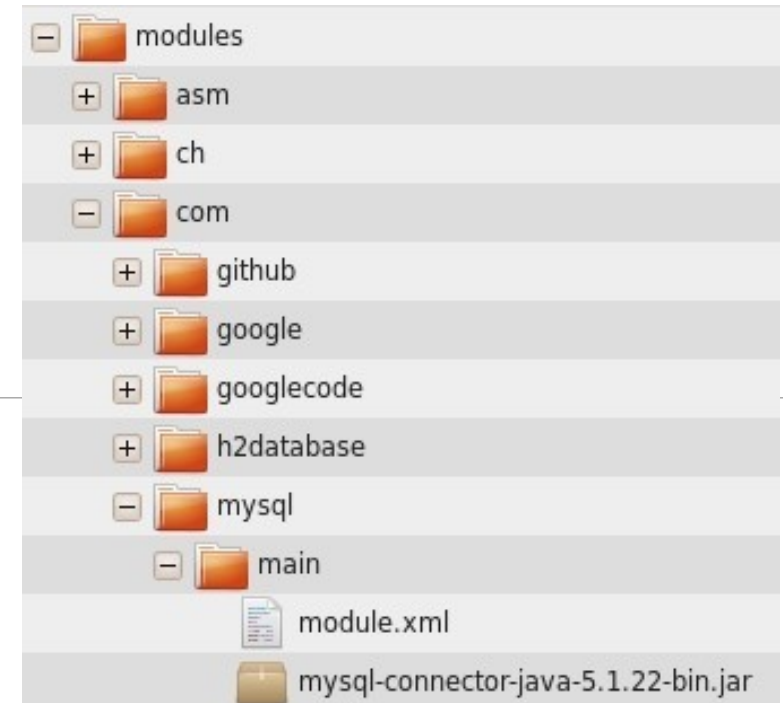
- Depuis JBoss 7, l'installation d'un driver JDBC est possible de deux façons :
 - ▶ en utilisant la notion de modules de JBoss 7
 - ▶ c'est le mode explicitement privilégié
 - ▶ nécessite la création d'un module JBoss manuellement et sa copie sur tous les hôtes du domaine
 - ▶ en déployant le JAR du driver comme une application classique
 - ▶ procédure simple,
 - ▶ permet notamment la diffusion du JAR vers chacun des hôtes mais :
 - ▶ *problèmes avec certains drivers*
 - ▶ *informations manquantes pour utiliser la notion de DataSource XA*

Installation d'un driver JDBC (2)

- Pour installer un driver sous forme de module :

- ▶ placer le fichier jar dans une arborescence sous `<JBOSS_HOME>/modules/`
 - ▶ ex : `com.mysql`
 - ▶ créer un sous répertoire 'main' pour y mettre le JAR et le fichier `module.xml`
- ▶ créer le fichier `module.xml`
 - ▶ l'attribut 'name' doit correspondre à l'arborescence

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.0" name="com.mysql">
  <resources>
    <resource-root path="mysql-connector-java-5.1.22-bin.jar"/>
  </resources>
  <dependencies>
    <module name="javax.api"/>
    <module name="javax.transaction.api"/>
  </dependencies>
</module>
```



Installation d'un driver JDBC (3)

- Une fois le driver installé sous forme de module, il faut le déclarer dans le profil :

▶ sous-système 'jboss:admin:datasources'

```
<subsystem xmlns="urn:jboss:domain:datasources:1.1">
  <datasources>
    <datasource jndi-name="java:jboss/datasources/ExampleDS" pool-name="ExampleDS" enabled="true" use-java-context="true">
      <connection-url>jdbc:h2:mem:test;DB_CLOSE_DELAY=-1</connection-url>
      <driver>h2</driver>
      <security>
        <user-name>sa</user-name>
        <password>sa</password>
      </security>
    </datasource>
  </datasources>
  <drivers>
    <driver name="h2" module="com.h2database.h2">
      <xa-datasource-class>org.h2.jdbcx.JdbcDataSource</xa-datasource-class>
    </driver>
    <driver name="mysql" module="com.mysql">
      <xa-datasource-class> com.mysql.jdbc.jdbc2.optional.MysqlXADataSource
    </xa-datasource-class>
    </driver>
  </drivers>
</datasources>
</subsystem>
```

Configuration d'une DataSource (1)

- Le sous-système 'jboss:admin:datasources' contient aussi les définitions de DataSource :

```
<subsystem xmlns="urn:jboss:domain:datasources:1.1">
  <datasources>
    <datasource jta="false" jndi-name="java:/ds/TestMySQL" pool-name="TestMySQL" enabled="false" use-ccm="false">
      <connection-url>jdbc:mysql://192.168.0.49:3306/test</connection-url>
      <driver-class>com.mysql.jdbc.Driver</driver-class>
      <driver>mysql</driver>
      <pool>
        <min-pool-size>0</min-pool-size>
        <max-pool-size>10</max-pool-size>
        <prefill>true</prefill>
      </pool>
      <security>
        <user-name>root</user-name>
        <password>admin</password>
      </security>
      <validation>
        <validate-on-match>false</validate-on-match>
        <background-validation>false</background-validation>
      </validation>
      <statement>
        <share-prepared-statements>false</share-prepared-statements>
      </statement>
    </datasource>
  </datasources>
</subsystem>
```

Configuration d'une DataSource (2)

- Ajout d'une DataSource par la console :

The screenshot illustrates the process of adding a new DataSource in the JBoss Enterprise Application Platform 6.0 console. The main interface shows the 'Profile' tab with 'Datasources' selected in the left sidebar. The 'Available Datasources' table lists 'ExampleDS' with a 'JNDI' of 'java:jboss/datasources/ExampleDS' and an 'Enabled?' checkbox. To the right of this table are buttons for 'Add', 'Remove', and 'Disable', with the number '3' indicating the 'Add' button. Below the main interface, three numbered steps of the 'Create Datasource' wizard are shown:

- Step 1/3: Datasource Attributes** (labeled 4): Shows fields for 'Name' (TestMySQL) and 'JNDI Name' (java:/ds/TestMySQL). A 'Next >>' button is at the bottom.
- Step 2/3: JDBC Driver** (labeled 5): Shows a list of drivers with 'postgresql-9.1-902.jdbc4.jar' selected. A 'Next >>' button is at the bottom.
- Step 3/3: Connection Settings** (labeled 6): Shows fields for 'Connection URL' (jdbc:mysql://localhost:3306/Test), 'Username' (admin), 'Password' (****), and 'Security Domain'. A 'Done' button is at the bottom.

Additional annotations include '1' pointing to the 'Profile' tab, '2' pointing to the 'Datasources' sidebar item, and 'Activation/Désactivation' pointing to the 'Enabled?' checkbox in the 'Available Datasources' table.

Obligation d'utiliser le préfixe 'java:/'

Configuration d'une DataSource (3)

- La liste des DataSources permet :

- ▶ d'activer/désactiver une DataSource
- ▶ de modifier sa configuration
- ▶ de tester la DataSource

The screenshot displays the JBoss Enterprise Application Platform 6.0 configuration console. The left sidebar shows the navigation tree with 'Databases' selected. The main panel shows the 'Databases' configuration page, specifically the 'JDBC Datasources' section. A table lists available datasources, including 'ExampleDS' and 'TestMySQL'. The 'TestMySQL' datasource is selected, and its configuration is shown in the 'Selection' tab. The 'Test Connection' button is highlighted. A 'Datasource Connection' dialog box is open in the top right corner, displaying a success message: 'Successfully created JDBC connection. Successfully connected to database TestMySQL.'.

1 Profile Runtime

2 Databases

3 Add Remove Disable

Name	JNDI	Enabled?
ExampleDS	java:jboss/datasources/ExampleDS	✓
TestMySQL	java:/ds/TestMySQL	✓

4 Test Connection

5 Edit

Connection URL: jdbc:mysql://192.168.0.49:3306/test New Connection Sql:

Need Help?

- **Fichiers -ds.xml :**

- ▶ dans les versions précédentes de JBoss AS la déclaration de chaque DataSource se faisait dans un fichier spécifique
- ▶ ce mode de déploiement reste supporté
- ▶ mais déconseillé explicitement pour l'exploitation

6.1.5. Deployment of -ds.xml files

In JBoss Enterprise Application Platform 6, datasources are defined as a resource of the server subsystem. In previous versions, a *** -ds.xml** datasource configuration file was required in the deployment directory of the server configuration. *** -ds.xml** files can still be deployed in JBoss Enterprise Application Platform 6, following the schema available here:

http://docs.jboss.org/ironjacamar/schema/datasources_1_1.xsd.



Warning

This feature should only be used for development. It is not recommended for production environments, because it is not supported by the JBoss administrative and management tools.

EAP 6 – Administration and Configuration Guide – section 6.1.5

Référence de DataSource (1)

- **La notion de DataSource permet d'isoler l'application des paramètres spécifiques à la base de données :**
 - ▶ mais le nom JNDI utilisé dans le code de l'application et celui défini par l'administrateur doivent être identiques
- **La notion de référence permet de créer un niveau d'indirection :**
 - ▶ Le développeur déclare l'existence d'une référence vers une DataSource dans le fichier web.xml. Il donne un nom à la référence et l'utilise dans le code

```
<resource-ref id="MaRef">  
  <res-ref-name>MaReferenceDeDataSource</res-ref-name>  
  <res-type>javax.sql.DataSource</res-type>  
  ...
```

- ▶ Lors du déploiement, l'administrateur doit associer cette référence à une DataSource existante
- ▶ Avec JBoss, l'association se fait dans le fichier jboss-web.xml :
 - ▶ ce qui diminue largement l'intérêt de cette notion

Référence de DataSource (2)

- **Ouverture d'une connexion :**

```
Context ctx = new InitialContext();  
DataSource ds = (DataSource)ctx.lookup(dsName);  
Connection connexion = ds.getConnection();
```

- **Sans utilisation de la référence :**

```
dsName = "java:/ds/MaDataSource"
```

- **Si utilisation de la référence :**

```
dsName = "java:comp/env/MaReferenceDeDataSource"
```

- **Fichier jboss-web.xml**

```
<?xml version="1.0" encoding="UTF-8"?>  
<resource-ref>  
  <res-ref-name>MaReferenceDeDataSource</res-ref-name>  
  <jndi-name>java:/ds/MaDataSource</jndi-name>  
</resource-ref>
```

Tracer les appels JDBC

- Au niveau de chaque DataSource, il est possible d'activer la trace de tous les appels JDBC :

- ▶ Modifier la définition de la DataSource :

```
<subsystem xmlns="urn:jboss:domain:datasources:1.1">
  <datasources>
    <datasource jta="false" jndi-name="java:/ds/TestMySQL"
      pool-name="TestMySQL" enabled="false" use-ccm="false" spy="true">
    ...
  </datasources>
</subsystem>
```

- ▶ Ajouter un logger :

- ▶ les traces apparaîtront dans le fichier de log du serveur

```
<logger category="jacorb.config">
  <level name="ERROR"/>
</logger>
<logger category="jboss.jdbc.spy">
  <level name="DEBUG"/>
</logger>
<root-logger>
  <level name="INFO"/>
  <handlers>
    <handler name="CONSOLE"/>
    <handler name="FILE"/>
  </handlers>
</root-logger>
```



Pool de connexion



Configuration du pool

La configuration du pool est optionnelle car JBoss applique des valeurs par défaut si rien n'est précisé.

Il est donc possible de préciser :

- ***min-pool-size*** : Le minimum de connexions dans le pool (défaut 0)
- ***max-pool-size*** : Le maximum de connexions dans le pool (défaut 20)
- ***prefill*** : booléen indiquant si le pool doit être initialisé avec le minimum
- ***use-strict-min*** : booléen indiquant que les connexions idle sont fermées afin de revenir au nombre minimal de connexions

```
<pool>
  <min-pool-size>5</min-pool-size>
  <max-pool-size>10</max-pool-size>
  <prefill>true</prefill>
  <use-strict-min>true</use-strict-min>
</pool>
```



Ajustement

Pour déterminer un dimensionnement correct, il est nécessaire de surveiller l'usage des connexions.

Cela peut être fait par l'interface CLI qui permet de visualiser les propriétés à l'exécution d'une source de données.

Une fois le pic de connexions trouvés, positionner le maximum à au moins 25-30% de plus

- Attention, si l'attribut *prefill* est utilisé, toutes les connexions seront marquées comme Active.



Example CLI

```
{xtypo_code}[standalone@localhost:9999
  /]/subsystem=datasources/data-source="java:/MySQLDS":read-resource(include-runtime=true)
{
  "outcome" => "success",
  "result" => {
    "ActiveCount" => "10",
    "AvailableCount" => "29",
    "AverageBlockingTime" => "0",
    "AverageCreationTime" => "56",
    "CreatedCount" => "10",
    "DestroyedCount" => "0",
    "MaxCreationTime" => "320",
    "MaxUsedCount" => "5",
    "MaxWaitCount" => "0",
    "MaxWaitTime" => "1",
    . . . .
  }
}
```



Autres outils de surveillance

D'autres outils sont disponibles. Par exemple, un simple client BD peut suffire :

- Oracle : Requête sur la vue *V\$SESSION*
- MySQL : *SHOW FULL PROCESSLIST*
- Postgre-SQL : Requête sur la table *PG_STAT_ACTIVITY*

L'outil P6Spy permet également d'accéder à ces données



Cache d'instruction

Le cache d'instruction s'applique pour les *prepared statement* ou *callable statement*

- Le cache est actif si la valeur de ***prepared-statement-cache-size*** est supérieure à 0
- La propriété ***track-statements*** à true autorise la fermeture automatique des statements et ResultSets
- ***share-prepared-statements*** détermine si deux requêtes dans la même transaction utilise le même statement

```
<statement>
```

```
<track-statements>true</track-statements>
```

```
<prepared-statement-cache-size>10</prepared-statement-cache-size>
```

```
<share-prepared-statements/>
```

```
</statement>
```



Timeout

La section `<timeout/>` contient :

- ***query-timeout*** : le timeout en secondes pour l'exécution d'une requête SQL,
- ***idle-timeout-minutes*** : le timeout en minutes avant qu'une connexion idle soit fermée. (Valeur par défaut 15 minutes, 0 pour désactiver)



xa-datasource

```
<datasources>
  <xa-datasource jndi-name="java:/XAMySqlDS" pool-name="MySqlDS_Pool"
enabled="true" use-java-context="true" use-ccm="true">
    <xa-datasource-property name="URL">jdbc:mysql://localhost:3306/MyDB</xa-
datasource-property>
    <xa-datasource-property name="User">jboss</xa-datasource-property>
    <xa-datasource-property name="Password">jboss</xa-datasource-property>
    <driver>mysql-xa</driver>
  </xa-datasource>
<drivers>
  <driver name="mysql-xa" module="com.mysql">
    <xa-datasource-class>com.mysql.jdbc.jdbc2.optional.MysqlXADataSource</xa-
datasource-class>
  </driver>
</drivers>
</datasources>
```



Mots de passe cryptés

Pour ne pas écrire le mot de passe en clair dans la configuration. Il faut créer un domaine de sécurité spécifique

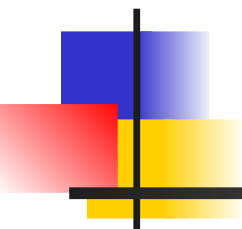
```
<security-domain name="EncryptedPassword">
  <authentication>
    <login-module code="SecureIdentityLogin" flag="required">
      <module-option name="username" value="test"/>
      <module-option name="password" value="encrypted_password"/>
    </login-module>
  </authentication>
</security-domain>
```

Puis le référencer dans la source de données :

```
<datasource ... >
  .....
  <security>
    <security-domain>EncryptedPassword</security-domain>
  </security>
</datasource>
```

Pour crypter le mot de passe :

```
java -cp $JBOSS_HOME/modules/org/picketbox/main/picketbox-4.0.6.<beta|final>.jar:$JBOSS_HOME/
modules/org/jboss/logging/main/jboss-logging-3.1.0.<some_version>.jar:$CLASSPATH
org.picketbox.datasource.security.SecureIdentityLoginModule password
```



2nd Level Cache



Cache de second-niveau

Avec la couche JPA, il est possible de définir un cache de second niveau dans Hibernate via *persistence.xml*

```
<shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
<properties>
  <property name="hibernate.cache.use_second_level_cache"
    value="true"/>
  <property name="hibernate.cache.use_minimal_puts" value="true"/>
  <!-- Optionnellement les statistiques -->
  <property name="hibernate.cache.infinispan.statistics"
    value="true"/>
</properties>
```



@Cacheable

Une fois configuré, il suffit d'annoter les entités que l'on veut cacher avec *@Cacheable*

Typiquement, on cache les entités qui ne sont accédés principalement qu'en lecture.



Métriques

Les principaux métriques accessible par JMX sont :

- **Evictions** : Nombre d'évictions
- **ReadWriteRatio** : Ratio lecture/écriture pour le cache
- **NumberofEntries** : Nombre d'entrée dans le cache
- **TimeSinceReset** : Durée en secondes depuis la dernière réinitialisation
- **ElapsedTime** : Durée en secondes depuis le démarrage du cache
- **Hits** : Nombre d'accès au cache
- **Misses** : Nombre d'accès manqués
- **HitRatio** : Pourcentage hit/(hit+miss)
- **AverageWriteTime, AverageReadTime** : Tps moyens d'accès



Merci!!!

❖ MERCI DE VOTRE ATTENTION



Annexes



ThreadPool

Les pools de Thread traite 2 problématiques :

- Ils apportent des gains de performance lors qu'ils exécutent un grand nombre de tâches asynchrones grâce à la pré-crédation de thread
- Ils fournissent un moyen de limiter et gérér les ressources nécessaires à l'exécution de tâches



Configuration

Chaque sous-système de JBoss utilisant des pools de threads gère leur propre configuration

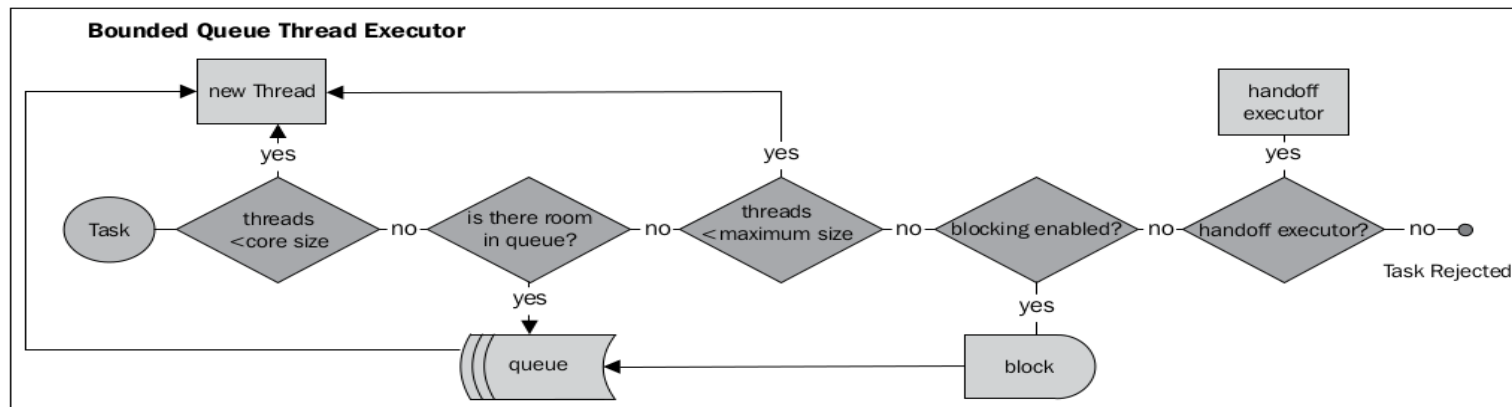
La configuration inclut

- La configuration de l'usine à Thread (configuration avancée)
- La configuration des pools **limité**
- La configuration des pools **illimité**
- La configuration des pools **sans file d'attente**
- La configuration du pool des threads **planifiées**

Pools de threads limité

Un pool de thread limité définit des contraintes sur la taille du pool via 3 attributs principaux :

- **core-threads** : La taille cœur du pool inférieure à la taille maximale
- **max-threads** : La taille maximale du pool
- **queue-length** : La taille de la file d'attente pour l'exécution d'une tâche





Exemple

<bounded-queue-thread-pool

name="jca-short-running" blocking="true">

<core-threads count="10" per-cpu="20"/>

<queue-length count="10" per-cpu="20"/>

<max-threads count="10" per-cpu="20"/>

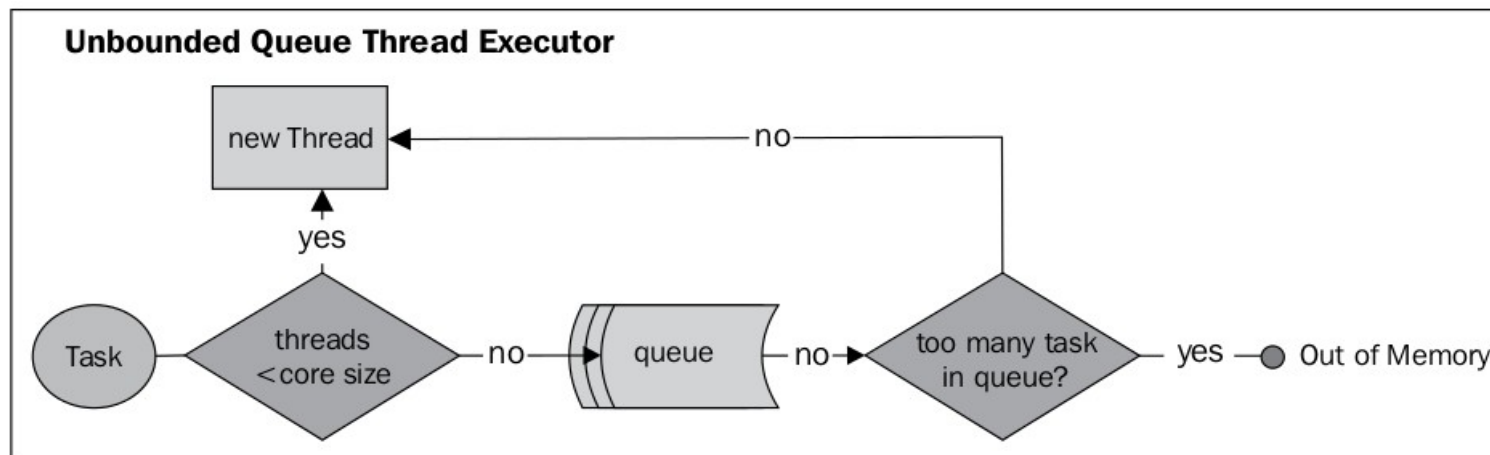
<keepalive-time time="10" unit="seconds"/>

</bounded-queue-thread-pool>

Pools de threads illimités

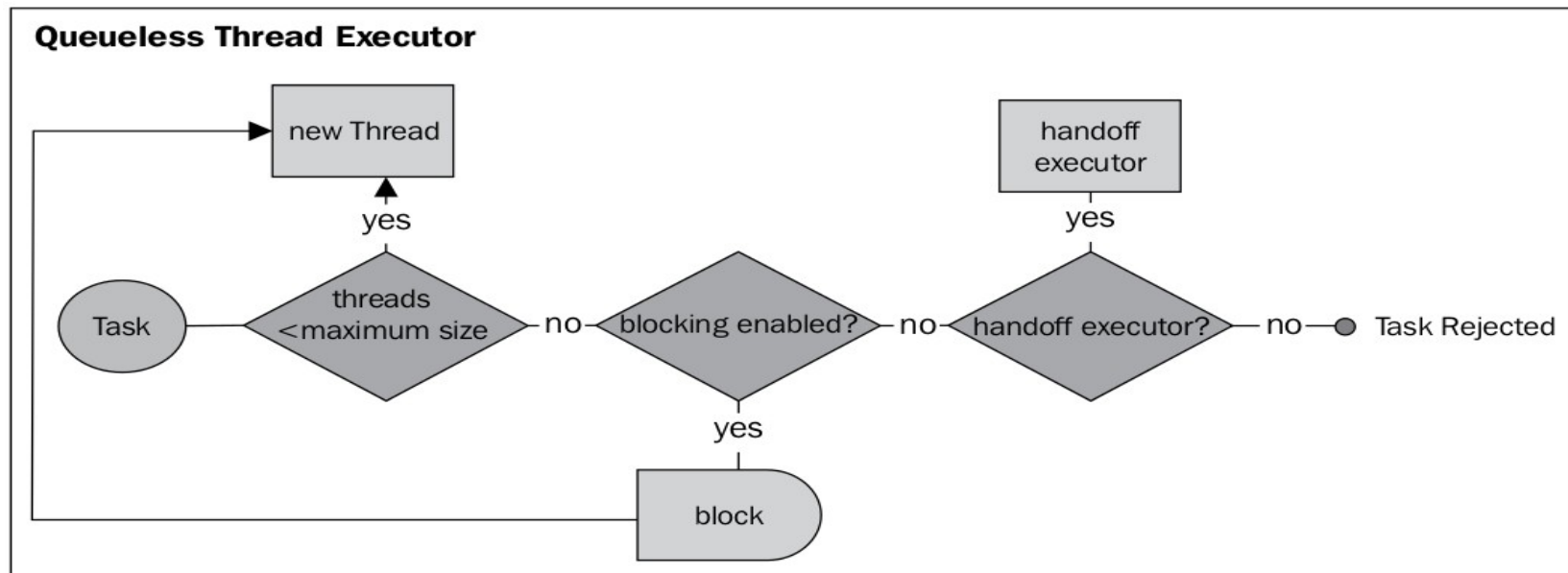
Ce type de pool accepte tout le temps des tâches.

La taille de la file d'attente n'est pas limitée et peut donc provoquer des *OutOfMemory*



Pools sans file d'attente

Le pool a une taille maximale. Lorsque la taille maximale est atteinte, soit la taille est rejetée, soit l'appelant est bloqué.



Pools programmés

Ces pools sont utilisés pour les tâches périodiques avec ou sans délai.

Ce type de pool est utilisé par les systèmes de remoting et de messaging JMS (*HornetQ*)

