Java Performance

Monitorización y análisis

Performance

Performance :: Intro

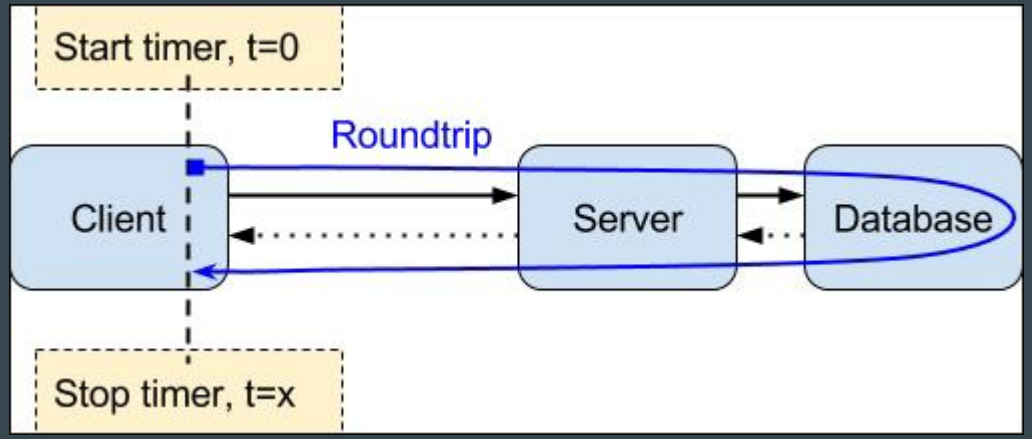
En el mundo de la IT, el rendimiento se utiliza a menudo como un término genérico de medida. Esta medida puede ser experimentado de manera algo diferente dependiendo de la función una persona tiene.

Un usuario de una determinada aplicación pensará más o menos favorable de la aplicación en función de la rapidez con que responde, o los flujos, desde su punto de vista y las interacciones individuales. Para un desarrollador, administrador, o alguna otra persona con una visión más técnica de la aplicación, el rendimiento puede significar varias cosas, por lo que tendrá que ser definido y cuantificado en más detalle. Estos roles perito deberá distinguir entre el tiempo de respuesta, rendimiento y eficiencia de utilización de recursos.

Performance :: Intro (cont)

Un ejemplo típico, (representado en el siguiente diagrama), es un usuario que ha llenado un formulario en una página web. Cuando el usuario envía el formulario haciendo clic en el Submit botón y envía los datos en el formulario, el temporizador se pone en marcha. Como los datos son recibidos por un servidor, los datos se rellena un JavaBean en un servlet de Java. Desde el servlet, se producirán posteriores llamadas a otros componentes tales como otros servlets y EJB. Algunos datos serán persistieron en una base de datos. Otros datos pueden ser recuperados de la misma u otras bases de datos, y todo lo que se transformarán en un nuevo conjunto de datos en forma de una página HTML que se envía de vuelta al navegador del usuario final. Como esta respuesta de los datos se materializa en el extremo del usuario, el temporizador se detiene y el tiempo de respuesta de la ida y vuelta puede ser revelado:

Request Response



Tools

jps :: Java Virtual Machine Process Status Tool

jinfo :: Java Information Tool

jcmd :: Java command Tool

Memory Management

Heap, Stack y Garbage Collectors

Heap :: Intro

El *heap* es un área de memoria que, de acuerdo con la especificación, no tiene que ser contigua, pero a menudo lo es en la implementación. El *heap* se crea durante el inicio de JVM y es compartido por todos los subprocesos de JVM. Su tamaño puede ser estático pero también puede crecer a un tamaño determinado según las necesidades de la aplicación en ejecución. En la clase Heap, las instancias y matrices se almacenan. Por lo tanto, esta área de memoria a menudo se denota como el almacenamiento de datos reales de JVM.

Esto se debe principalmente al hecho de que las estructuras de datos, con algunas restricciones, pueden asignarse en cualquier posición de memoria dentro del *heap*.

Ver el documento JVM memory management with the GC

Heap :: Intro

Además, como estas estructuras de datos ya no están activas, la memoria que asignaron será liberada o recuperada tarde o temprano, lo que provocará la fragmentación del almacenamiento dinámico.

Dentro del heap, hay varias áreas de memoria separadas cuyos tamaños dependen del tamaño de la cantidad de memoria disponible en el *heap* por defecto. Estas áreas se denominan áreas de memoria generacional y se nombran de la siguiente manera:

- Young generation
- Old generation (o tenured)
- Permanent generation (PermGen)

Young Generation

Los objetos Java residen en un área llamada *heap*. Este se crea cuando la JVM se inicia y puede aumentar o disminuir de tamaño mientras se ejecuta la aplicación. Cuando el heap se llena, se "recoge la basura" mediante el *Garbage Collector*. Durante este proceso, los objetos que ya no se utilizan se borran, lo que deja espacio para nuevos objetos.

Tenga en cuenta que la JVM usa más memoria que solo el heap. Por ejemplo, los métodos Java, las pilas de subprocesos y los identificadores nativos se asignan en memoria separada del heap, así como las estructuras de datos internas de JVM.

El *heap* a veces se divide en dos áreas (o *generation*) llamadas *young generation* y el *old generation*.

Young Generation (cont)

El área *young* es una parte del *heap* reservado para la asignación de objetos nuevos. Cuando el *young generation* se llena, se recoge la basura con una colección especial para jóvenes, donde todos los objetos que han vivido lo suficiente en el vivero se promueven (mueven) al viejo espacio, liberando así el vivero para una mayor asignación de objetos. Cuando el espacio antiguo se llena, la basura se recoge allí, un proceso llamado colección anterior.

El razonamiento detrás de la *young generation* es que la mayoría de los objetos son temporales y de corta duración, por lo cual está diseñada para ser veloz en encontrar objetos recién asignados que aún están vivos y alejarlos de esta. Este proceso está diseñado para liberar memoria de manera mucho más rápida.

Young Generation :: Motivación

Después de la memoria total disponible, el segundo factor más influyente que afecta el rendimiento del Garbage Collector es la proporción del heap dedicado a la *young generation*. Cuanto mayor es esta, menos *GC minor* ocurren. Sin embargo, para un *heap size* limitado, una *young generation* más grande implica un *old generation* más pequeña, lo que aumentará la frecuencia de las GC major. However, for a bounded heap size, a larger young generation implies a smaller tenured generation, which will increase the frequency of major collections. La elección óptima depende de la distribución del ciclo de vida de los objetos asignados por la aplicación.

Young Generation :: Configuración

Podemos configurar el tamaño de la memoria young dependiendo de la aplicación dando así mejores respuestas a aplicaciones que rindan mejor necesitando este tipo de memoria.

-Xmn<young size>[g|m|k]

Young Generation :: Áreas

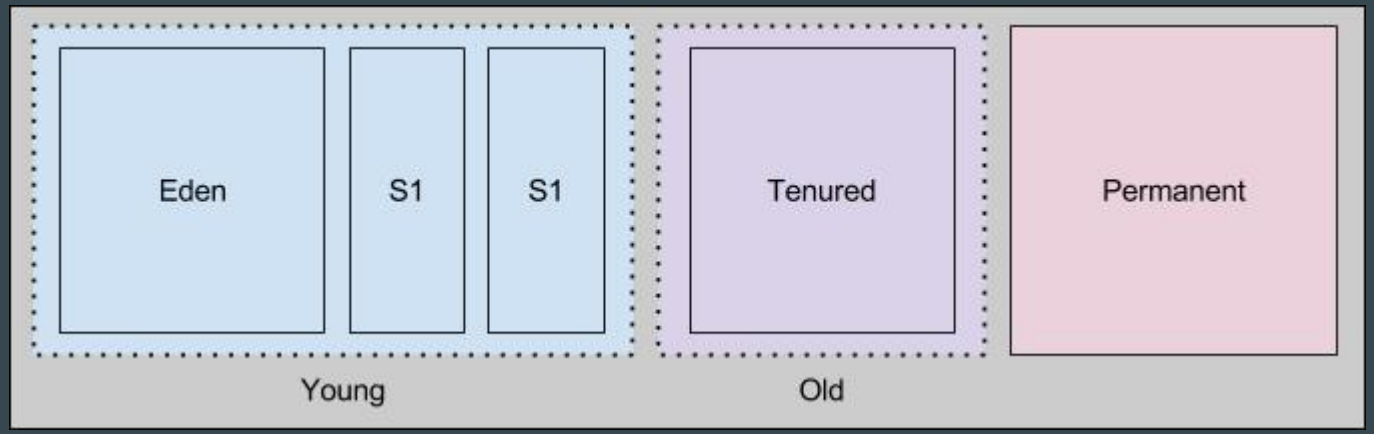
Esta se divide en tres área

- Eden
- Survivor space 0 (S0)
- Survivor space 1 (S1)

Las estructuras de datos se almacenan en las diferentes áreas de memoria de la *young generation* y en la *permanent generation*, según cuánto tiempo hayan estado vivos. Aquí, la vida se define en el número de colecciones realizadas por el *garbage collector* de JVM. De hecho, la razón principal para tener todas estas áreas de memoria es optimizar el rendimiento del GC y, por lo tanto, el uso de la memoria.

Permanent Generation

En la *PermGen*, se almacenan las estructuras de datos y la metainformación de nuestras clases.



J8, de Permgen a Metaspace

La principal diferencia entre PermGen y Metaspace reside desde la perspectiva del usuario es que el Metaspace, de forma automática, aumenta automáticamente su tamaño (hasta lo que proporciona el sistema operativo), mientras que PermGen siempre tiene un tamaño máximo fijo. Puede establecer un máximo fijo para Metaspace con parámetros JVM, pero no puede hacer que PermGen aumente automáticamente.

En gran medida, es solo un cambio de nombre. Antes, cuando se introdujo PermGen, no había Java EE o carga y descarga dinámica del classloader, por lo que una vez que se cargaba una clase, se quedaba atascada en la memoria hasta que se apagaba la JVM. Hoy en día las clases pueden cargarse y descargarse durante la vida útil de la JVM, por lo que Metaspace tiene más sentido para el área donde se guardan los metadatos.

J8, de Permgen a Metaspace (cont)

Ambos contienen las instancias de java.lang.Class y ambos sufren leaks de ClassLoader. La única diferencia es que con las configuraciones predeterminadas de Metaspace, toma más tiempo hasta que nota los síntomas (ya que aumenta automáticamente tanto como puede), es decir, simplemente aleja el problema sin resolverlo. OTOH Imagino que el efecto de quedarse sin memoria OS puede ser más grave que simplemente quedarse sin JVM PermGen, así que no estoy seguro de que sea una gran mejora.

Ya sea que esté utilizando una JVM con PermGen o con Metaspace, si está realizando una descarga de clase dinámica, debe tomar medidas contra las fugas del classloader.

-XX:MaxMetaspaceSize=<metaspace size>[g|m|k]

Garbage Collection Overview

Young generation collectors

Hay varios tipos de colectores diferentes de la GC, que responde a estrategias diferentes implementaciones dependiendo del fabricante y necesidad. Hoy en día, existen muchas implementaciones diferentes y estos son mezclas de las estrategias básicas. En primer lugar, vamos a echar un vistazo a la mayoría estrategias comunes del Hotspot VM:

- El serial collector
- El parallel collector
- El colector concurrente
- La Garbage First (G1)

Old generation collectors

Garbage Collection : Algoritmos

Algoritmos

Hay varios Garbage Collectors disponibles en las JVM actuales:

- Serial Collector
- Throughput Parallel Collector,
- Parallel Old GC (Parallel Compacting GC)
- Concurrent Mark & Sweep GC (or "CMS")
- Garbage First - G1.

Sus características de rendimiento son bastante diferentes, sin embargo, comparten conceptos básicos

Serial GC (-XX:+UseSerialGC)

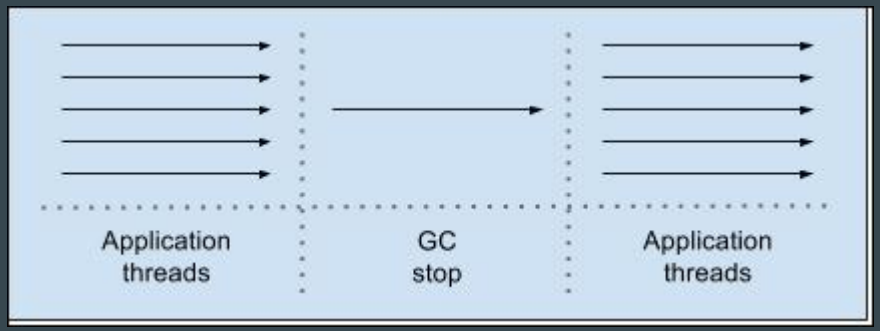
Como se puede ver, el colector de serie no es realmente una opción para las soluciones empresariales de hoy en día. En caso de que su aplicación puede implementar en una máquina multiprocesador y requieren para completar el mayor número posible de transacciones en una ventana de tiempo, el colector paralelo es una buena opción. Este es el caso de las aplicaciones que realizan actividades de procesamiento por lotes.

Tener procesadores rápidos y una aplicación que necesita para servir a cada solicitud por una cantidad de tiempo estricta es generalmente un caso para el colector concurrentes. El colector concurrente es particularmente adecuado para aplicaciones que tienen un relativamente gran conjunto de datos de larga vida, ya que puede recuperar objetos mayores sin una larga pausa. Este es generalmente el caso en las aplicaciones web donde se almacena una cantidad consistente de la memoria en el HttpSession.

Parallel GC (-XX:+UseParallelGC)

Serial collector (-XX:+UseSerialGC)

El serial collector utiliza un solo hilo como se muestra en el siguiente diagrama. Este hilo se detiene todos los otros hilos, una JVM llamado comportamiento de parada-el-mundo. Si bien es un colector relativamente eficiente, ya que no hay sobrecarga de comunicación entre los hilos, no puede tomar ventaja de las máquinas de multiprocesador. Por lo tanto, es el más adecuado para un solo procesador máquinas y como sabemos, estos ya no son comunes.



Parallel Old GC(-XX:+UseParallelOldGC)

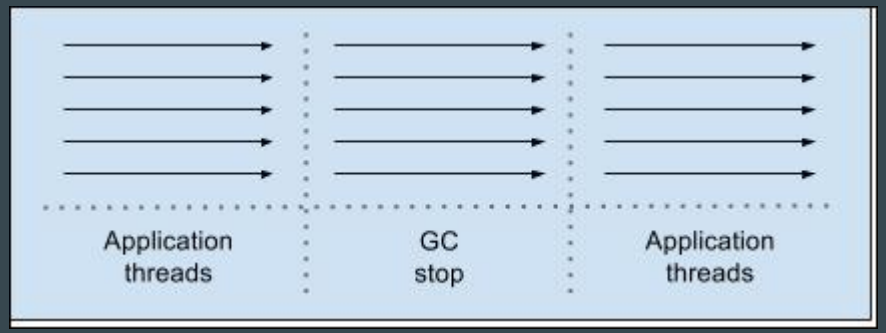
El colector paralelo, también conocido comúnmente como el colector de rendimiento, es el colector por defecto en las máquinas de grado servidor para Java sE 7. también se puede activar de forma explícita con el siguiente parámetro VM:

-XX:+UseParallelGC

El colector paralelo realiza colecciones de menor importancia en paralelo, lo que puede reducir significativamente mejorar el rendimiento de las aplicaciones que tienen un heap de colecciones menores.

Parallel Old GC(cont)

Como se puede ver en el siguiente diagrama, el colector paralelo todavía requiere una denominada stop-the-world. actividad Sin embargo, puesto que las colecciones se realizan en paralelo, haciendo buen uso de muchas CPU, disminuye la sobrecarga del garbage collector y por lo tanto aumenta el rendimiento de la aplicación.



CMS GC (-XX:+UseConcMarkSweepGC)

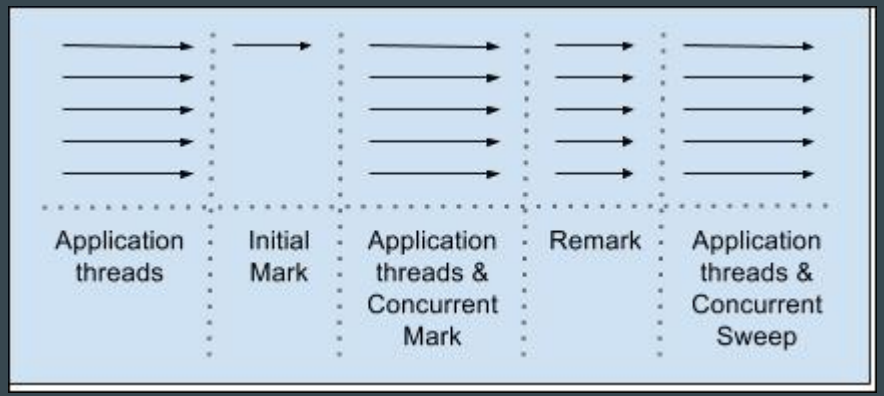
El colector concurrente es un colector de baja pausa más comúnmente conocido como Concurrent Mark Sweep (CMS) Se realiza la mayor parte de su trabajo al mismo tiempo que la aplicación sigue ejecutando. Esto optimiza el rendimiento manteniendo pausas GC corto.

Básicamente, este colector consume recursos del procesador con el fin de tener mayores tiempos de pausa colección más cortos. Esto puede suceder debido a que el colector concurrente utiliza un solo hilo garbage collector que se ejecuta simultáneamente con las roscas de la aplicación. Por lo tanto, el propósito del colector es concurrente para completar la colección de la generación titular antes de que se llene.

CMS GC (cont)

Básicamente, este colector consume recursos del procesador con el fin de tener tiempos de pausa colección más cortos. Esto puede suceder debido a que el colector concurrente utiliza un solo hilo garbage collector que se ejecuta simultáneamente con las roscas de la aplicación. Por lo tanto, el propósito del colector es concurrente para completar la colección de la generación titular antes de que se llene.

El siguiente diagrama nos da una idea de cómo funciona el colector concurrentes:



G1 Young Generation (-XX:+UseG1GC)

Si quieres entender G1 GC, olvida todo lo que sabes sobre la generación joven y la generación anterior. Como puede ver en la imagen, un objeto se asigna a cada cuadrícula, y luego se ejecuta un GC. Luego, una vez que un área está llena, los objetos se asignan a otra área, y luego se ejecuta un GC. Los pasos donde los datos se mueven desde los tres espacios de la generación joven a la generación anterior no se pueden encontrar en este tipo de GC. Este tipo fue creado para reemplazar el CMS GC, que ha causado muchos problemas y quejas a largo plazo. La mayor ventaja del GC G1 es su rendimiento. Es más rápido que cualquier otro tipo de GC.

G1 GC :: Motivos

Se sabe que un GC puede romper los SLA de una aplicación mediante, una pausa imprevista del Garbage Collector prolongada puede exceder fácilmente los requisitos de tiempo de respuesta de una aplicación que de otro modo funciona. Además, la irregularidad aumenta cuando tiene un GC no compactante como Concurrent Mark and Sweep (CMS) que intenta reclamar su heap fragmentado con una recolección de basura completa en serie (de un único subproceso) que es stop-the-world (STW).

Como? Supongamos que una falla de asignación en la generación *young* desencadena una recolección, lo que lleva a promocionarse a la *old*. Además, supongamos que la generación *old* fragmentada tiene espacio insuficiente para los objetos recién promovidos. Tales condiciones desencadenarían un ciclo completo de recolección, que realizará la compactación del *heap*.

G1 GC

El colector G1 está incluido y totalmente compatible-en el Oracle JDK 7 actualización 4 distribución de Java SE 7. G1 y está dirigido a entornos de servidores con CPUs multinúcleo equipadas con grandes cantidades de memoria. Se llama un colector parallel-concurrent regionalizado y se habilita mediante el uso de la -XX:+UseG1GC.

Cuando se utiliza G1, el heap se divide en regiones de igual tamaño entre 1 y 32 MB. JVM establece este tamaño en el arranque. El objetivo es tener no más de 2048 regiones en una máquina virtual. El Eden (E), Survivor (S),y Tenured (T)de la generación se dividen en conjuntos lógicos no continuas de estas regiones, como se visualiza en el diagrama siguiente:

Combinaciones de Garbage Collectors

- -XX:+UseSerialGC : *young* Copy y *old* MarkSweepCompact
- -XX:+UseG1GC : *young* G1 Young y *old* G1 Mixed
- -XX:+UseParallelGC -XX:+UseParallelOldGC -XX:+UseAdaptiveSizePolicy : *young* PS Scavenge *old* PS MarkSweep con tamaño adaptativo
- -XX:+UseParallelGC -XX:+UseParallelOldGC -XX:-UseAdaptiveSizePolicy : *young* PS Scavenge *old* PS MarkSweep, sin tamaño adaptativo
- -XX:+UseParNewGC (deprecated) *young* ParNew *old* MarkSweepCompact
- -XX:+UseConcMarkSweepGC -XX:+UseParNewGC : *young* ParNew *old* ConcurrentMarkSweep**
- -XX:+UseConcMarkSweepGC -XX:-UseParNewGC (deprecated) *young Copy*, old ConcurrentMarkSweep

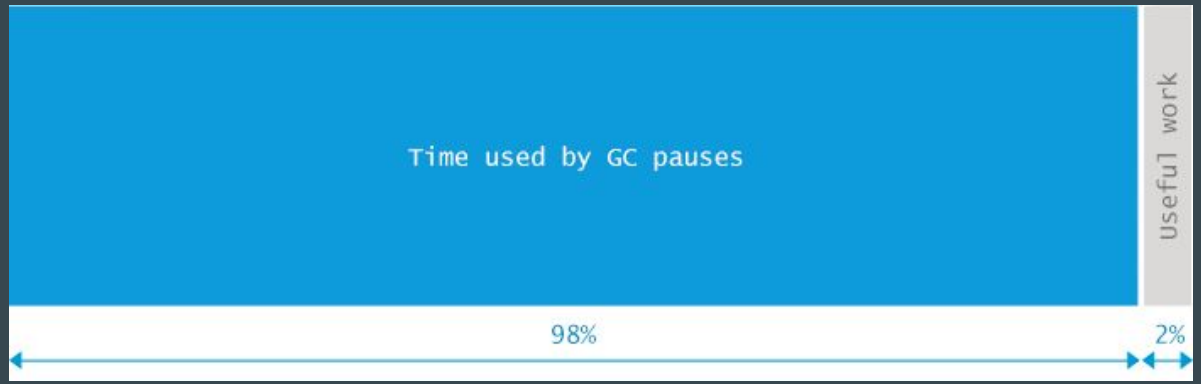
Combinaciones de Garbage Collectors

- -XX:+/-CMSIncrementalMode (deprecated) - habilita o deshabilita el algoritmo incremental.
- -XX:+/-CMSConcurrentMTEnabled - habilita o deshabilita algoritmo paralelo concurrente (multiple threads)
- -XX:+/-UseCMSCompactAtFullCollection - habilita o deshabilita la compactación con el *full GC* se ejecuta

Garbage Collector :: Overhead

De forma predeterminada, la JVM está configurada para emitir este error si gasta más del 98% del tiempo total de GC y cuando después del GC solo se recupera menos del 2% del heap. ... Tenga en cuenta que el java.lang.OutOfMemoryError: GC overhead limit exceeded solo se produce cuando el 2% de la memoria se libera después de varios ciclos de GC.

-XX:SurvivorRatio=<ratio>



Garbage Collector :: Analizando

La JVM nos permite trazar y analizar el comportamiento del GC mediante la escritura en logs de datos que a posterior pueden ser aggregados.

```
-XX:+PrintGCTimeStamps -Xloggc:<logfile>  -XX:+PrintGCDetails
```

Para ello podemos usar la herramienta https://github.com/chewiebug/GCViewer/wiki/Features

En caso de necesitar rotación de logs indique los siguientes flags para evitar ficheros gigantes.

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

Comparando GC

Heap Análisis

...

Obtener dump del Heap

Heap Dump :: Obtener

Los dump del heap nos permiten analizar la memoria en la máquina virtual mediante herramientas o flags de la JVM.

- jmap
- jstack
- HeapDumpOnOutOfMemoryError
- HeapDumpOnCtrlBreak
- -agentlib:hprof=heap=dump,format=b

jmap:

32 bit JVM:

```
jmap -dump:format=b,file=<heap_dump_filename> <pid>
```

64 bit JVM:

```
jmap -J-d64 -dump:format=b,file=<heap_dump_filename> <pid>
```

64 bit JVM con G1GC (Solamente objetos “vivos” del heap):

```
jmap -J-d64 -dump:live,format=b,file=<heap_dump_filename> <pid>
```

jstack ::

-XX:HeapDumpOnOutOfMemoryError

Es un flag muy utilizado para obtener un dump al darse un error de OutOfMemoryException (OOM)

Heap Dump :: Análisis

Heap dumps pueden ser capturados con interfaces graficas como jvisualvm eclipse memory analyzer, yourkit o mission control; así como mediante línea de comando con jcmd o jmap.

Stack

Stack

Una pila (*stack)* es un tipo de almacenamiento del tipo *last in, first out* (LIFO). Para cada subproceso de ejecución de JVM, hay una pila de JVM. En esta pila, las entradas llamadas cuadros se almacenan. Los marcos pueden contener referencias a objetos, valores de variables y resultados parciales. Durante la ejecución de una aplicación Java, estos marcos se agregan (push) a, o se eliminan (pop) de la pila de JVM.

En JVM, también existe el concepto de una pila nativa. Normalmente, existe una pila por subproceso de JVM, y se utiliza para admitir funciones / métodos nativos (escritos en un lenguaje nativo de plataforma como C / C ++) ya que las pilas de JVM normales no pueden contenerlos.

Stack vs Heap :: Intro

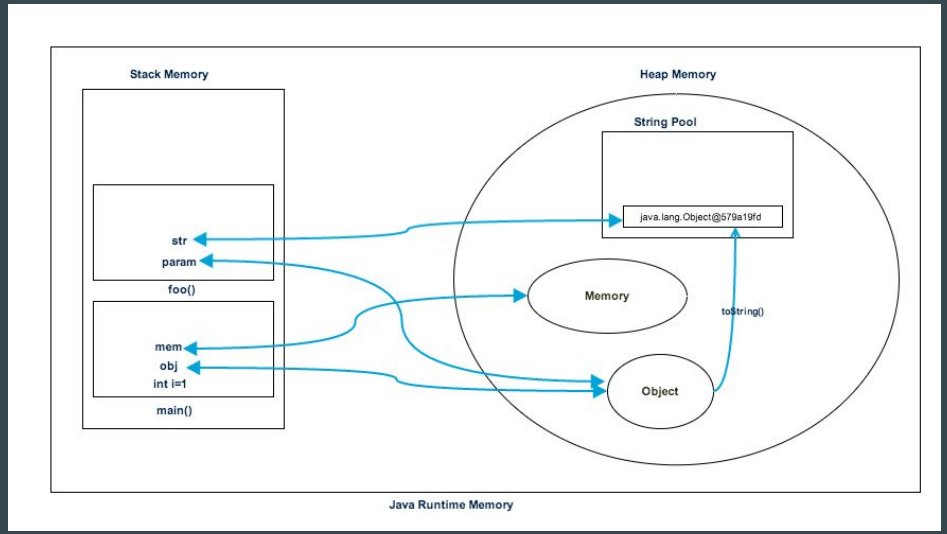
El espacio de Java *heap* es utilizado por java runtime para asignar memoria a clases de objetos y JRE. Cada vez que creamos un objeto, siempre se crea en el espacio de *heap*.

Garbage Collection se ejecuta en la memoria del *heap* para liberar la memoria utilizada por los objetos que no tienen ninguna referencia. Cualquier objeto creado en el espacio de almacenamiento dinámico tiene acceso global y se puede referenciar desde cualquier lugar de la aplicación.

La memoria Java Stack se usa para la ejecución de un hilo. Contienen valores específicos del método que son efímeros y referencias a otros objetos en el heap que se derivan del método.

Stack vs Heap

- La memoria heap es utilizada por todas las partes de la aplicación, mientras que la memoria de pila se usa solo por un hilo de ejecución.
- Cada vez que se crea un objeto, siempre se almacena en el espacio *heap* y la memoria contiene la referencia al mismo. La memoria *stack* sólo contiene variables primitivas locales y variables de referencia para objetos en el *heap*.
- Los objetos almacenados en el *heap* son accesibles globalmente mientras que otros hilos no pueden acceder a la memoria del *stack*.



Stack vs Heap

- La administración de la memoria en la pila se realiza de manera LIFO, mientras que es más compleja en la memoria *heap* porque se usa globalmente. La memoria *heap* se divide en *young generation* , old *generation*, etc.
- La memoria *stack* es de corta duración, mientras que la memoria *heap* vive desde el inicio hasta el final de la ejecución de la aplicación.
- Podemos usar la opción -Xms y -Xmx JVM para definir el tamaño de inicio y el tamaño máximo de la memoria *heap*. Podemos usar -Xss para definir el tamaño de la memoria *stack*.
- Cuando la memoria *stack* está llena, java arroja *StackOverFlowError* mientras que si la memoria del *heap* está llena, arroja OutOfMemoryError con el mensaje *"Java Heap Space"* .
- El tamaño del *stack* es muy inferior en comparación con la memoria *heap*, debido a la simplicidad en la asignación de memoria (LIFO), la memoria del *stack* es muy rápida en comparación con la memoria del *heap*.

Otros recursos

Java Pass By Value Stack Heap Memory Explanation (YouTube)

Comprehensive Java Developer's Guide (DZone)

Java's Garbage-Collected Heap (artima developer)

Java Heap and Stack (Guru99)

JVM Memory Management (JavaBeat)

Shallow vs. Retained Heap

Shallow vs. Retained Heap

El *shallow heap* es la memoria consumida por un objeto. Un objeto necesita 32 o 64 bits (según la arquitectura del sistema operativo) por referencia, 4 bytes por entero, 8 bytes por largo, etc. Dependiendo del formato de volcado del *heap*, el tamaño puede ajustarse (por ejemplo, alineado a 8, etc.) para modelar mejor el consumo real de la máquina virtual.

El *retained set* de X es el conjunto de objetos que serían eliminados por GC cuando X es basura.

El *retained heap* de X es la suma de los tamaños superficiales de todos los objetos en el conjunto retenido de X, es decir, la memoria que X mantiene viva.

Shallow vs. Retained Heap

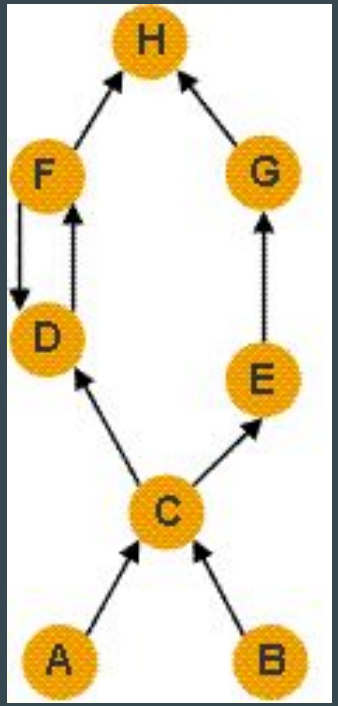
En términos generales, un *shallow heap* de un objeto es su tamaño en el heap y el tamaño retenido del mismo objeto es la cantidad de memoria del *heap* que se liberará cuando el objeto sea recolectado.

El *retained set* para un conjunto principal de objetos, como todos los objetos de una clase en particular o todos los objetos de todas las clases cargadas por un classloader particular o simplemente un conjunto de objetos arbitrarios, es el conjunto de objetos que se libera si todos los objetos de ese conjunto principal se vuelve inaccesible. El *retained set* incluye estos objetos, así como todos los demás objetos a los que solo se puede acceder a través de estos objetos. El *retained size* es el tamaño de almacenamiento dinámico total de todos los objetos contenidos en el retained set.

Shallow vs. Retained Heap

- **A** y B son GC Roots
- El tamaño mínimo retenido proporciona una estimación buena (inferior) del tamaño retenido que se calcula de manera más rápida que el tamaño retenido exacto de un conjunto de objetos. Solo depende del número de objetos en el conjunto inspeccionado, no del número de objetos en el heap dump.

Leading Set	Retained Set
E	E,G
C	C,D,E,F,G,H
A,B	A,B,C,D,E,F,G,H



Links y recursos

- https://github.com/google/allocation-instrumenter (google-allocation-instumenter)
- http://blog.javabenchmark.org/2013/07/compute-java-object-memory-footprint-at.html (with JAMM)
- https://github.com/jbellis/jamm (JAMM src)

Cómo determinar el tamaño de los objetos

- http://www.javaworld.com/article/2074458/core-java/estimating-java-object-sizes-with-instrumentation.html (guia corta DYI)
- https://www.youtube.com/results?search_query=Understanding+Java+GC (webCast on how GC traverses objects for similar purposes)

Garbage Collection Roots

Garbage Collection Roots

Un root garbage collection es un objeto al que se puede acceder desde fuera del heap. Las siguientes razones hacen que un objeto sea una raíz de GC:

Clase del sistema
Clase cargada por el cargador de clase de arranque / sistema. Por ejemplo, todo desde rt.jar como java.util. *.
JNI Local
Variable local en código nativo, como código JNI definido por el usuario o código interno JVM.
JNI Global
Variable global en código nativo, como código JNI definido por el usuario o código interno JVM.
Bloque de hilo
Objeto al que se hace referencia desde un bloque de subprocesos actualmente activo.

Garbage Collection Roots (cont)

Hilo

Un hilo iniciado, pero no detenido.

Monitor ocupado

Todo lo que ha llamado wait () o notify () o que está sincronizado. Por ejemplo, llamando a sincronizado (Objeto) o ingresando un método sincronizado. El método estático significa clase, método no estático significa objeto.

Java Local

Variable local. Por ejemplo, parámetros de entrada u objetos de métodos creados localmente que todavía están en la pila de un hilo.

Pila nativa

Parámetros de entrada o salida en el código nativo, como el código JNI definido por el usuario o el código interno de JVM. Este es a menudo el caso ya que muchos métodos tienen partes nativas y los objetos manejados como parámetros del método se convierten en raíces de GC. Por ejemplo, los parámetros utilizados para los métodos de E / S de archivos / redes o la reflexión.

Garbage Collection Roots (cont)

Finalizable

Un objeto que está en una cola esperando que se ejecute su finalizador.

Sin finalizar

Un objeto que tiene un método de finalización, pero que no se ha finalizado y aún no se encuentra en la cola del finalizador.

Inalcanzable

Un objeto que es inalcanzable desde cualquier otra raíz, pero ha sido marcado como raíz por MAT para retener objetos que de otro modo no se incluirían en el análisis.

Marco de pila de Java

Un marco de pila Java, que contiene variables locales. Solo se genera cuando el volcado se analiza con la preferencia establecida para tratar los marcos de la pila de Java como objetos.

Desconocido

Un objeto de tipo raíz desconocido.

Memory Leaks

...

Introducción

Memory leaks :: Intro

Un problema de rendimiento muy común en el desarrollo de software es la pérdida de memoria. El lenguaje Java ha hecho mucho para minimizar este problema mediante el *garbage collector* liberando objetos sin referencias, por ejemplo, pero un código deficiente puede y aún creará problemas.

Cuando ocurre un OOME y se sospecha que el problema se debe a una fuga de memoria, es hora de comenzar a investigar la (s) causa (s) real (es).

Memory leaks :: Detection

Un proceso simple pero efectivo para encontrar fugas de memoria se puede definir en los siguientes tres pasos:

- Encuentra instancias de clase sospechosas de fuga
- Encuentre dónde se han instanciado
- Encuentre el motivo (s) por el que existe la fuga

Memory leaks :: Detection

La regla de oro para identificar una posible fuga de *heap* es la siguiente:

Si el uso del tamaño de almacenamiento dinámico sigue aumentando durante un tiempo cada vez que se ejecuta un GC completo, implica una posible pérdida de memoria. Para obtener la mejor estimación, el mismo tipo y cantidad de operaciones se deben realizar iterativamente durante cada ciclo antes / después de los GC completos.

Al usar una herramienta como VisualVM, puede, en algunos casos obvios, ser posible identificar una fuga directamente simplemente observando el gráfico de la memoria del espacio dinámico. Si sigue creciendo a pesar de que se están ejecutando los GC completos, es hora de investigar.

ver documento

PermGen/Metaspace Memory Leaks :: Detection

Hay muchas razones por las que su aplicación puede encontrarse con un `java.lang.OutOfMemoryError: PermGen` space. La causa fundamental de la mayoría de ellos es alguna referencia a un objeto o una clase cargada por el Classloader de la aplicación que ha muerto después de eso. O un enlace directo al Classloader. Las acciones que debe tomar para el remedio son bastante similares para la mayoría de los casos de fuga. En primer lugar, averiguar dónde se llevará a cabo esa referencia. En segundo lugar, añadir un shutdown hook para eliminar la referencia durante undeployment. Usted puede hacer eso, ya sea usando un listener, servlet o mediante el uso de la API.

Ver documento PermGen Memory Leak

Tools

...

herramientas para la monitorización y análisis de memoria

jmap - Memory Map

jmap -heap

jmap -histo

```
Usage:
        jmap [option] <pid>
        (to connect to running process)
        jmap [option] <executable <core>
        (to connect to a core file)
        jmap [option] [server_id@]<remote server IP or hostname>
        (to connect to remote debug server)

where <option> is one of:
        <none>           to print same info as Solaris pmap
        -heap            to print java heap summary
        -histo[:live]    to print histogram of java object heap; if the "live"
                         suboption is specified, only count live objects
        -clstats         to print class loader statistics
        -finalizerinfo   to print information on objects awaiting finalization
        -dump:<dump-options> to dump java heap in hprof binary format
                         dump-options:
                         live             dump only live objects; if not specified,
                                          all objects in the heap are dumped.
                         format=b         binary format
                         file=<file>  dump heap to <file>
                         Example: jmap -dump:live,format=b,file=heap.bin <pid>
        -F               force. Use with -dump:<dump-options> <pid> or -histo
                         to force a heap dump or histogram when <pid> does not
                         respond. The "live" suboption is not supported
                         in this mode.
        -h | -help       to print this help message
        -J<flag>         to pass <flag> directly to the runtime system
```

jmcd ::

- JFR.stop
- JFR.start
- JFR.dump
- JFR.check
- VM.native_memory
- VM.check_commercial_features
- VM.unlock_commercial_features
- ManagementAgent.stop
- ManagementAgent.start_local
- ManagementAgent.start
- GC.rotate_log
- GC.class_stats
- GC.class_stats
- GC.class_histogram
- GC.heap_dump
- GC.run_finalization
- GC.run
- Thread.print
- VM.uptime
- VM.flags
- VM.system_properties
- VM.command_line
- VM.version
- help

jstat :: Java Status Tool

Podemos usar la herramienta de línea de comandos jstat para monitorear la memoria JVM y las actividades del *Garbage Collector*.

S0C	S1C	S0U	S1U	EC	EU	OC	OU	MC	MU	CCSC	CCSU	YGC	YGCT	FGC	FGCT	GCT
28160.0	29696.0	0.0	0.0	522752.0	56330.9	106496.0	21033.5	37632.0	36716.3	4352.0	4116.0	8	0.252	3	0.414	0.666

- S0C y S1C: tamaño de las áreas Survivor0 y Survivor1 (kb).
- S0U y S1U: uso de las áreas Survivor0 y Survivor1 (kb). Siempre vacías.
- CE y UE: tamaño y uso del espacio *Eden* (kb). El tamaño de la UE está aumentando y tan pronto como cruza la CE, se llama *GC minor* y se reduce el tamaño de la UE.

jstat :: Java Status Tool

- OC y OU: tamaño y el uso de la *old generation* (kb).
- PC y PU: tamaño y el uso actual de *Perm Gen* (kb).
- YGC y YGCT: cantidad de eventos del GC ocurridos en la *young generation y* tiempo acumulado para las operaciones de GC. (ambos se incrementan en la misma fila donde se descarta el valor de EU debido a *GC minor*.
- FGC y FGCT: cantidad y tiempo acumulado de eventos del *GC full*. Tenga en cuenta que el tiempo total de *GC* es demasiado alto en comparación con los tiempos de *young generation*.
- GCT: tiempo total acumulado para las operaciones del *GC (YGCT + FGCT*).

jmc :: Java Mission Control

Oracle Java Mission Control es una herramienta disponible en Oracle JDK desde Java 7u40. Esta herramienta proviene de JRockit JVM, donde estuvo disponible durante años. JRockit y su versión de JMC fueron bien descritos en Oracle JRockit: The Definitive Guidebook escrito por dos desarrolladores senior de JRockit (También visite el blog de Marcus Hirt, el primer lugar donde debería buscar cualquier noticia de JMC).

Oracle JMC podría usarse con 2 propósitos principales:

- Monitorizar el estado de múltiples JVM Oracle en ejecución
- Análisis de archivo de volcado de Java Flight Recorder

jvisualvm :: VisualVM

VisualVM es una herramienta que ha evolucionado para dar soporte a varias áreas interesantes relacionadas con el ajuste. En general y out-of-the-box, VisualVM normalmente proporciona las siguientes cinco características principales que se visualizan en diferentes vistas en la interfaz gráfica.

Descripción general: una vista de información no interactiva sobre la información del núcleo de JVM que contiene argumentos, propiedades y algo de supervisión.

Subprocesos: se pueden tomar volcados de subprocesos y los subprocesos (activos o finalizados) de la JVM se visualizan en diferentes formatos (línea de tiempo, tabla y detalles).

jvisualvm :: VisualVM

Monitor: se pueden pedir volcados de pila y GC completos. Además, gráficos en vivo están en exhibición para los siguientes componentes:

- Uso de CPU y actividad de GC
- Uso de memoria Heap o PermGen con valores de contador del tamaño utilizado y tamaño máximo para ambas áreas de memoria
- Clases con valores de contador para carga, descarga y carga / descarga compartida
- Hilos con valores de contador para live, daemon, started y peak

Profiler: el generador de perfiles admite tanto el perfil de rendimiento (CPU) como la memoria.

jvisualvm :: VisualVM

Sampler: la muestra tiene una configuración y funcionalidad similar a la del generador de perfiles. Últimamente, el desarrollo, sin embargo, se ha centrado en la muestra, y ha recibido más información sobre PermGen y los hilos en comparación con el generador de perfiles, por ejemplo.

Eclipse Memory Analyzer (MAT)

Es una gran herramienta de análisis de memoria para determinar consumos, leaks.

ver memory leaks

Buscando Leaks con MAT

Memory Leaks : Finding Memory Leak with Eclipse Memory Analyzer

JVM

...

Flags y configuración

Xmx2048M -Xms2048M -XX:ParallelGCThreads=8 -Xincgc
-XX:+UseConcMarkSweepGC -XX:+UseParNewGC
-XX:+CMSIncrementalPacing
-XX:+AggressiveOpts -XX:+CMSParallelRemarkEnabled
-XX:+DisableExplicitGC -XX:MaxGCPauseMillis=500
-XX:SurvivorRatio=16
-XX:TargetSurvivorRatio=90 -XX:+UseAdaptiveGCBoundary
-XX:-UseGCOverheadLimit -Xnoclassgc -XX:UseSSE=3 -XX:PermSize=128m
-XX:LargePageSizeInBytes=4m

Para que?

Configurando JVM

Introducción

Existen más de 680 opciones para configurar la JVM, muchas de ellas se relacionan con debugging, performance y behavior.

Podemos investigar las misma mediante

```
java -XX:+PrintFlagsFinal -version
```

Behaviors

Behaviors Flags

-XX:-AllowUserSignalHandlers No se queje si la aplicación instala manejadores de señal. (Solo relevante para Solaris y Linux).

-XX:AltStackSize = 16384 Tamaño de pila de señal alternativa (en Kbytes). (Relevante sólo para Solaris, eliminado de 5.0)

-XX:-DisableExplicitGC Por defecto, las llamadas a System.gc() están habilitadas (-XX:-DisableExplicitGC).

Use -XX:+DisableExplicitGC para deshabilitar las llamadas a System.gc().

Tenga en cuenta que la JVM aún realiza la recolección de elementos no utilizados cuando es necesario.

XX:+FailOverToOldVerifier Falló al verificador antiguo cuando falla el nuevo verificador de tipos. (Introducido en 6.)

-XX:+HandlePromotionFailure La colección de la generación más joven no requiere una garantía de promoción completa de todos los objetos en vivo. (Introducido en 1.4.2 actualización 11) [5.0 y anterior: falso.]

-XX:+MaxFDLimit Aumenta el número de descriptores de archivos al maximo. (Relevante sólo para Solaris).

Behaviors Flags

-XX: PreBlockSpin = 10 Variable de Spin Count para usar con -XX:+UseSpinning. Controla las iteraciones de giro máximas permitidas antes de ingresar el código de sincronización del hilo del sistema operativo. (Introducido en 1.4.2)

-XX: -RelaxAccessControlCheck Relaja las verificaciones de control de acceso en el verificador. (Introducido en 6.)

-XX:+ScavengeBeforeFullGC Hace GC de generación joven antes de un GC completo. (Introducido en 1.4.1.)

-XX:+UseAltSigs Use señales alternativas en lugar de SIGUSR1 y SIGUSR2 para señales internas de VM. (Introducido en la actualización 1.3.1 9, 1.4.1. Relevante solo para Solaris).

-XX:+UseBoundThreads Vincula los hilos del nivel de usuario con los hilos del kernel. (Relevante solo para Solaris).

-XX:-UseConcMarkSweepGC Usa una colección simultánea de barrido de marcas para la generación anterior. (Introducido en 1.4.1)

Behaviors Flags

-XX:+UseGCOverheadLimit Utilice una política que limite la proporción de tiempo de la VM que se gasta en GC antes de que se produzca un error de OutOfMemory.

-XX:+UseLWPSincronización Utilice la sincronización basada en LWP en lugar de la basada en subprocesos. (Introducido en 1.4.0. Relevante solo para Solaris).

-XX:-UseParallelGC Usa recolección de basura paralela para carroña. (Introducido en 1.4.1)

-XX:-UseParallelOldGC Usa recolección de basura paralela para las colecciones completas. Habilitar esta opción establece automáticamente -XX:+UseParallelGC. (Introducido en la actualización 5.0 5.0)

-XX:-UseSerialGC Usa la recolección de basura en serie. (Introducido en 5.0)

-XX:-UseSpinning Habilita el giro ingenuo en el monitor Java antes de ingresar el código de sincronización del hilo del sistema operativo. (Solo relevante para 1.4.2 y 5.0.) [1.4.2, plataformas de Windows multiprocesador: verdadero]

Behaviors Flags

-XX:+UseTLAB Utilice la asignación de objetos locales de subprocesos (Introducido en 1.4.0, conocido como UseTLE antes de eso.) [1.4.2 y anteriores, x86 o con -cliente: falso]

-XX:+UseSplitVerifier Use el nuevo verificador de tipos con atributos StackMapTable. (Introducido en 5.0.) [5.0: falso]

-XX:+UseThreadPriorities Usa prioridades de subprocesos nativos.

-XX:+UseVMInterruptibleIO La interrupción de subprocesos antes o con EINTR para operaciones de E/S da como resultado OS_INTRPT.

-XX:ExitOnOutOfMemoryError y -XX:CrashOnOutOfMemoryError

Desde 8u92, la JVM sumo dos flags nuevos : ExitOnOutOfMemoryError y CrashOnOutOfMemoryError.

-XX:ExitOnOutOfMemoryError : Cuando habilitamos esta opción, la JVM aborta la ejecución a la primera excepción out-of-memory. Puede ser utilizado en caso de desear reiniciar la instancia en lugar de esperar que la JVM repare o gestione el error.

-XX:CrashOnOutOfMemoryError : Si esta opción está habilitada, la JVM, cuando un out-of-memory ocurra, la JVM hace un crash produciendo un texto y un fichero binario.

Performance

Performance Flags

-XX:+AggressiveOpts Activa las optimizaciones del compilador de rendimiento de punto que se espera sean predeterminadas en las próximas versiones. (Introducido en la actualización 5.0 5.0)

-XX: CompileThreshold = 10000 Número de invocaciones de método / branches antes de compilar [-client: 1,500]

-XX: LargePageSizeInBytes = 4m Establece el tamaño de página grande utilizado para el heap de Java.

-XX: MaxHeapFreeRatio = 70 Porcentaje máximo de pila libre después de GC para evitar la reducción.

-XX: MaxNewSize = tamaño Tamaño máximo de nueva generación (en bytes). Desde 1.4, MaxNewSize se calcula como una función de NewRatio. [1.3.1 Sparc: 32m; 1.3.1 x86: 2.5m.]

-XX: MaxPermSize = 64m Tamaño de la generación permanente. [5.0 y versiones posteriores: las máquinas virtuales de 64 bits se escalaron un 30% más; 1,4 amd64: 96 m; 1.3.1 -cliente: 32m.]

-XX: MinHeapFreeRatio = 40 Porcentaje mínimo de pila libre después de GC para evitar la expansión.

Performance Flags

-XX: NewRatio = 2 Proporción de tamaños de generación antiguos / nuevos. [Sparc -cliente: 8; servidor x86: 8; x86 -cliente: 12.] - cliente: 4 (1.3) 8 (1.3.1+), x86: 12]

-XX: NewSize = 2m Tamaño predeterminado de la nueva generación (en bytes) [5.0 y versiones posteriores: las máquinas virtuales de 64 bits se escalaron un 30% más; x86: 1 m; x86, 5.0 y más: 640k]

-XX: ReservedCodeCacheSize = 32m Tamaño de caché de código reservado (en bytes) - tamaño máximo de caché de código.

-XX: SurvivorRatio = 8 Relación del tamaño del espacio eden / survivor.

-XX: TargetSurvivorRatio = 50 Porcentaje deseado de espacio de sobreviviente utilizado después del barrido.

-XX: ThreadStackSize = 512 Tamaño de pila de subprocesos (en Kbytes). (0 significa usar el tamaño de pila predeterminado)

-XX:+UseBiasedLocking Habilita el bloqueo sesgado. Para más detalles, mira este ejemplo de ajuste.

Performance Flags

-XX:+UseFastAccessorMethods Utilice versiones optimizadas del campo Get <Primitive>.

-XX: -UseISM Utilice la memoria compartida íntima. [No se acepta para plataformas que no son de Solaris.] Para obtener más información, consulte Memoria compartida íntima.

-XX:+UseLargePages Usa la memoria de página grande.

-XX:+UseMPSS Utilice soporte de tamaño de página múltiple con páginas de 4mb para el heap. No lo use con ISM ya que esto reemplaza la necesidad de ISM.

-XX:+UseStringCache Habilita el almacenamiento en caché de cadenas comúnmente asignadas.

-XX: AllocatePrefetchLines = 1 Número de líneas de caché que se cargarán después de la última asignación de objetos utilizando las instrucciones de captación previa generadas en el código compilado JIT. Los valores predeterminados son 1 si el último objeto asignado fue una instancia y 3 si fue una matriz.

-XX: AllocatePrefetchStyle = 1 Estilo de código generado para las instrucciones de captación previa.

Performance Flags

-XX:+UseCompressedStrings Use un byte [] para cadenas que se pueden representar como ASCII puro. (Introducido en Java 6 Update 21 Performance Release)

-XX:+OptimizeStringConcat Optimizar las operaciones de concatenación de cadenas siempre que sea posible.

G1 Garbage Collector

Garbage First (G1) :: Fases

El ciclo de marcado tiene las siguientes fases:

Fase de marca inicial: el G1 GC marca las raíces durante esta fase. Esta fase se lleva a cuestas en una recolección de basura joven normal (STW).

Fase de exploración de la región raíz: el GC G1 escanea las regiones supervivientes de la marca inicial para referencias a la generación anterior y marca los objetos referenciados. Esta fase se ejecuta simultáneamente con la aplicación (no STW) y debe completarse antes de que pueda comenzar la próxima colección de basura joven STW.

Fase de marca concurrente: el GC G1 encuentra objetos alcanzables (en vivo) en todo el heap. Esta fase ocurre al mismo tiempo que la aplicación, y puede ser interrumpida por las colecciones de basura jóvenes de STW.

Garbage First (G1) :: Fases

Fase de observación: esta fase es la recopilación de STW y ayuda a completar el ciclo de marcado. El GC G1 drena los almacenamientos intermedios SATB, rastrea los objetos vivos no visitados y realiza el procesamiento de referencia.

Fase de limpieza: en esta fase final, el GC G1 realiza las operaciones STW de depuración contabilidad y RSet. Durante la contabilidad, el G1 GC identifica regiones completamente libres y candidatos mixtos de recolección de basura. La fase de limpieza es parcialmente concurrente cuando se restablece y devuelve las regiones vacías a la lista libre.

Garbage First (G1) Flags

-XX:+UseG1GC Usa el Garbage First (G1)

-XX: MaxGCPauseMillis = n Establece un objetivo para el tiempo máximo de pausa del GC. Este es un objetivo suave, y la JVM hará su mejor esfuerzo para lograrlo.

-XX: InitiatingHeapOccupancyPercent = n Porcentaje de la ocupación de heap (completa) para iniciar un ciclo de GC concurrente. Es utilizado por los GC que activan un ciclo de GC concurrente basado en la ocupación de todo el heap, no solo de una de las generaciones (por ejemplo, G1). Un valor de 0 indica "hacer ciclos constantes de GC". El valor predeterminado es 45.

-XX: NewRatio = n Proporción de tamaños de generación antiguos / nuevos. El valor predeterminado es 2.

-XX: SurvivorRatio = n Relación del tamaño del espacio eden / survivor. (default 8)

-XX: MaxTenuringThreshold = n Valor máximo para el umbral de ocupación. (default 15).

-XX: ParallelGCThreads = n Establece el número de subprocesos utilizados durante las fases paralelas de los recolectores de basura.

Garbage First (G1) Flags (cont)

-XX: ConcGCThreads = n Número de subprocesos que utilizarán los garbage collectors concurrentes. El valor predeterminado varía según la plataforma en la que se ejecuta la JVM.

-XX: G1ReservePercent = n Establece la cantidad de heap que se reserva como falso techo para reducir la posibilidad de fallas de promoción. El valor predeterminado es 10.

-XX: G1HeapRegionSize = n Con G1, el heap se subdivide en regiones de tamaño uniforme. Esto establece el tamaño de las subdivisiones individuales. El valor predeterminado de este parámetro se determina ergonómicamente en función del tamaño del heap. El valor mínimo es 1Mb y el valor máximo es 32Mb.

-XX:+UseLargePages

Las CPU modernas tienen una característica llamada páginas de memoria grandes. Esta característica permite que las aplicaciones que requieren mucha memoria realicen asignaciones de 2 a 4 MB en lugar de las 4 KB estándar. Huelga decir que esto puede mejorar el rendimiento significativamente para algunas aplicaciones. Sin embargo, también puede causar un rendimiento degradado cuando la escasez de memoria (que a menudo ocurre en sistemas con un tiempo de actividad prolongado, donde la memoria se ha fragmentado tanto que las nuevas reservas de memoria no están permitidas) lleva a una paginación excesiva.

Memory

-XX: MaxPermSize: establece el tamaño máximo de espacio permanente (en bytes).
-XX: ThreadStackSize: establece el tamaño de la pila de subprocesos (en bytes).
-XX:+UseStringCache: habilita el almacenamiento en caché de cadenas comúnmente asignadas.
-XX: G1HeapRegionSize: establece el tamaño de subdivisión del heap G1 (en bytes).
-XX: MaxGCPauseMillis: establece un objetivo para el tiempo máximo de pausa del GC.
-XX: MaxNewSize: tamaño máximo de nueva generación (en bytes).
-XX:+AggressiveOpts: permite el uso de características agresivas de optimización del rendimiento.

Debugging

Debugging flags

-XX: -CITime imprime el tiempo invertido en el compilador JIT. (Introducido en 1.4.0)

-XX: ErrorFile =. / Hs_err_pid <pid> .log Si se produce un error, guarde los datos de error en este archivo. (Introducido en 6.)

-XX: -ExtendedDTraceProbes Habilita las sondas dtrace que afectan el rendimiento. (Introducido en 6. Relevante solo para Solaris).

-XX: HeapDumpPath =. / Java_pid <pid> .hprof Ruta al directorio o nombre de archivo para el volcado del heap.

-XX: -HeapDumpOnOutOfMemoryError Vaciar el heap al archivo cuando se lanza java.lang.OutOfMemoryError.

-XX: OnError = "<cmd args>; <cmd args>" Ejecuta comandos definidos por el usuario en caso de error fatal. (Introducido en la actualización 1.4.2 9.)

-XX: OnOutOfMemoryError = "<cmd args>;<cmd args>;" Ejecuta el(los) comando(s) definido(s) por el usuario cuando se lanza por primera vez OutOfMemoryError.

Debugging flags

-XX: -PrintClassHistogram Imprime un histograma de instancias de clase en Ctrl-Break. Manejable. (Introducido en 1.4.2.) El comando jmap -histo proporciona una funcionalidad equivalente.

-XX: -PrintConcurrentLocks Imprime bloqueos java.util.concurrent en el volcado de hilo Ctrl-Break.. (Introducido en 6.) El comando jstack -l proporciona una funcionalidad equivalente.

-XX: -PrintCommandLineFlags Imprimir banderas que aparecieron en la línea de comando.

-XX: -PrintCompilation Imprimir mensaje cuando se compila un método.

-XX: -PrintGC Imprimir mensajes en la recolección de basura. Manejable.

-XX: -PrintGCDetails Imprima más detalles en la recolección de basura.

-XX: -PrintGCTimeStamps Imprime las marcas de tiempo en la recolección de basura. Manejable

-XX: -PrintTenuringDistribution Imprime información de edad de tenencia.

Debugging flags

-XX: -PrintAdaptiveSizePolicy Permite la impresión de información sobre el tamaño de generación adaptativa.

-XX: -TraceClassLoading Carga de rastreo de clases.

-XX: -TraceClassLoadingPreorder Rastree todas las clases cargadas en el orden al que se hace referencia (no cargadas).

-XX: -TraceClassResolution Traza las resoluciones del conjunto constante.

-XX: -TraceClassUnloading Rastreo de descarga de clases.

-XX: -TraceLoaderConstraints Registro de seguimiento de las restricciones del cargador.

-XX:+PerfDataSaveToFile Guarda los datos binarios jvmstat al salir.

-XX: ParallelGCThreads = n Establece el número de subprocesos de recolección de basura en los recolectores de basura paralelos jóvenes y viejos. El valor predeterminado varía según la plataforma en la que se ejecuta la JVM.

Debugging flags

-XX:+UseCompressedOops Permite el uso de punteros comprimidos (referencias de objeto representadas como compensaciones de 32 bits en lugar de punteros de 64 bits) para un rendimiento optimizado de 64 bits con tamaños de almacenamiento Java inferiores a 32 gb.

-XX:+AlwaysPreTouch Toque previamente el heap de Java durante la inicialización de JVM. Por lo tanto, cada página del almacenamiento dinámico se pone a cero a demanda durante la inicialización en lugar de incrementalmente durante la ejecución de la aplicación.

-XX: AllocatePrefetchDistance = n Establece la distancia de captación previa para la asignación de objetos. La memoria que se va a escribir con el valor de los objetos nuevos se capta previamente en el caché a esta distancia (en bytes) más allá de la dirección del último objeto asignado. Cada subproceso de Java tiene su propio punto de asignación. El valor predeterminado varía según la plataforma en la que se ejecuta la JVM.

-XX: InlineSmallCode = n Inline un método previamente compilado solo si el tamaño del código nativo generado es menor que esto. El valor predeterminado varía según la plataforma en la que se ejecuta la JVM.

Debugging flags

-XX: MaxInlineSize = 35 Tamaño máximo de bytecode de un método para ser inline.

-XX: FreqInlineSize = n Tamaño máximo de bytecode de un método ejecutado frecuentemente para ser inline. El valor predeterminado varía según la plataforma en la que se ejecuta la JVM.

-XX: LoopUnrollLimit = n Desenrollar cuerpos de bucle con nodo de representación intermedia del compilador del servidor contar menos de este valor. El límite utilizado por el compilador del servidor es una función de este valor, no el valor real. El valor predeterminado varía según la plataforma en la que se ejecuta la JVM.

-XX: InitialTenuringThreshold = 7 Establece el umbral de ocupación inicial para usar en el dimensionamiento adaptivo de GC en el colector joven paralelo. El umbral de tenencia es la cantidad de veces que un objeto sobrevive a una colección joven antes de ser promovido a la generación anterior o titular.

-XX: MaxTenuringThreshold = n Establece el umbral máximo de permanencia para usar en el dimensionamiento adaptivo de GC. El valor máximo actual es 15. El valor predeterminado es 15 para el colector paralelo y 4 para CMS.

Debugging flags

-Xloggc: <nombre de archivo> Log de la salida detallada de la GC al archivo especificado. La salida detallada está controlada por los indicadores de GC verbose normales.

-XX: -UseGCLogFileRotation Habilitado la rotación del registro de GC, requiere -Xloggc.

-XX: NumberOfGClogFiles = 1 Establecer el número de archivos que se utilizarán al girar registros, debe ser> = 1. Los archivos de registro rotados utilizarán el siguiente esquema de nombres, <nombre de archivo> .0, <nombre de archivo> .1, ..., <nombre de archivo> .n-1.

-XX: GCLogFileSize = 8K El tamaño del archivo de registro en el que se rotará el registro debe ser> = 8K.

Server Flags y opciones

Java <8 Server Flags

- -server
- -Xms<heap size>[g|m|k] -Xmx<heap size>[g|m|k]
- -XX:PermSize=<perm gen size>[g|m|k]
- -XX:MaxPermSize=<perm gen size>[g|m|k]
- -Xmn<young size>[g|m|k]
- -XX:SurvivorRatio=<ratio>
- -XX:+UseConcMarkSweepGC
- -XX:+CMSParallelRemarkEnabled
- -XX:+UseCMSInitiatingOccupancyOnly
- -XX:CMSInitiatingOccupancyFraction=<percent>
- -XX:+ScavengeBeforeFullGC
- -XX:+CMSScavengeBeforeRemark
- -XX:+PrintGCDateStamps -verbose:gc
- -XX:+PrintGCDetails -Xloggc:"<path to log>"
- -XX:+UseGCLogFileRotation
- -XX:NumberOfGCLogFiles=10
- -XX:GCLogFileSize=100M
- -Dsun.net.inetaddr.ttl=<TTL in seconds>
- -XX:+HeapDumpOnOutOfMemoryError
- -XX:HeapDumpPath=<path to dump>`date`.hprof
- -Djava.rmi.server.hostname=<external IP>
- -Dcom.sun.management.jmxremote.port=<port>
- -Dcom.sun.management.jmxremote.authenticate=false
- -Dcom.sun.management.jmxremote.ssl=false

Java >=8 Server Flags

- -server
- -Xms<heap size>[g|m|k] -Xmx<heap size>[g|m|k]
- -XX:MaxMetaspaceSize=<metaspace size>[g|m|k]
- -Xmn<young size>[g|m|k]
- -XX:SurvivorRatio=<ratio>
- -XX:+UseConcMarkSweepGC
- -XX:+CMSParallelRemarkEnabled
- -XX:+UseCMSInitiatingOccupancyOnly
- -XX:CMSInitiatingOccupancyFraction=<percent>
- -XX:+ScavengeBeforeFullGC
- -XX:+CMSScavengeBeforeRemark
- -XX:+PrintGCDateStamps -verbose:gc
- -XX:+PrintGCDetails -Xloggc:"<path to log>"
- -XX:+UseGCLogFileRotation
- -XX:NumberOfGCLogFiles=10
- -XX:GCLogFileSize=100M
- -Dsun.net.inetaddr.ttl=<TTL in seconds>
- -XX:+HeapDumpOnOutOfMemoryError
- -XX:HeapDumpPath=<path to dump>`date`.hprof
- -Djava.rmi.server.hostname=<external IP>
- -Dcom.sun.management.jmxremote.port=<port>
- -Dcom.sun.management.jmxremote.authenticate=false
- -Dcom.sun.management.jmxremote.ssl=false

-XX:OnError

Mediante este flag podemos ejecutar comandos en caso de error.

Podremos enviar correos, ejecutar alarmas o limpiar ficheros.

-XX:HeapDumpOnOutOfMemoryError

Es un flag muy utilizado para poder determinar causas raíz analizando la memoria del

-XX:ExitOnOutOfMemoryError y -XX:CrashOnOutOfMemoryError

Desde 8u92, la JVM sumo dos flags nuevos : ExitOnOutOfMemoryError y CrashOnOutOfMemoryError.

-XX:ExitOnOutOfMemoryError : Cuando habilitamos esta opción, la JVM aborta la ejecución a la primera excepción out-of-memory. Puede ser utilizado en caso de desear reiniciar la instancia en lugar de esperar que la JVM repare o gestione el error.

-XX:CrashOnOutOfMemoryError : Si esta opción está habilitada, la JVM, cuando un out-of-memory ocurra, la JVM hace un crash produciendo un texto y un fichero binario.

Monitoring

...

VisualVM

VisualVM :: Características

VisualVM proporciona muchas funciones que son útiles para encontrar cuellos de botella y ajustar un sistema.

La herramienta puede tomar varios *snapshots* y *dumps* que se pueden ver directamente o guardar para un análisis posterior. Éstas incluyen:

Snapshot de la aplicación: incluye la información básica de JVM que, de lo contrario, está disponible en las vistas general y de monitor de VisualVM

Snapshot de CPU: esta instantánea enumera el uso del tiempo de ejecución del árbol de llamadas en el paquete / clase / nivel de método y puntos de acceso dentro de los nodos del árbol.

VisualVM :: Características

Snapshot de memoria: esta instantánea muestra los bytes, los objetos asignados y las generaciones por clase o tipo.

Dump de subprocesos: contiene una variedad de información de cada subproceso, como dirección, estado y prioridad, por mencionar algunos.

Dump de pila: contiene mucha información útil, como información básica de JVM como en la vista general, niveles de uso de clases e instancias con atributos y referencias, e hilos en el momento de la descarga.

Dump de VM Core: esto incluye un volcado de pila y de hilo como se describió anteriormente.

Java Mission Control

Java Flight Recorder

Java Flight Recorder es una función de Java Mission Control. Desde el punto de vista del usuario, se ejecuta con un tiempo de grabación fijo / máximo tamaño de archivo de grabación / longitud máxima de grabación (su aplicación puede finalizar antes) y espere hasta que se complete la grabación. Después de eso lo analizas en el Java Mission Control.

Para ejecutar agregar 2 opciones siguientes a la JVM a la que desea conectarse:

```
-XX:+UnlockCommercialFeatures -XX:+FlightRecorder
```

Cómo ejecutar Java Flight Recorder

Lo tercero a tener en cuenta es que, de forma predeterminada, la JVM permite hacer seguimientos de pila solo en puntos seguros. Como resultado, puede tener información de seguimiento de pila incorrecta en algunas situaciones. La documentación indica que configure 2 parámetros más si desea los seguimientos de pila más precisos (no podrá establecer esos parámetros en la JVM en ejecución):

```
-XX:+UnlockDiagnosticVMOptions -XX:+DebugNonSafepoints
```

Finalmente, si desea disponer de la mayor cantidad de E / S de archivos, excepciones de Java e información de perfiles de CPU, asegúrese de haber seleccionado los parámetros habilitados y sus umbrales establecidos en "1 ms".

Wildlfy

Tuning y monitoring

Wildfly JVM Flags

Wildfly Flags

Para el acceso remote JMX se requiere un usuario ManagementRealm y el flag que permite los accesos por ip

```
-Djboss.bind.address.management=<ip|hostname remote> (0.0.0.0 all)
```

Para ello, el cliente jmx utilizamos la cadena de control es

```
service:jmx:remoting-jmx://<hostname>:<port>

service:jmx:remoting-http://<hostname>:<port>
```

Configuración de la JVM

La configuración de los parámetros de JVM para que la mismo ejecute en WildFly se puede realizar de varias maneras. Las formas más comunes son las siguientes:

- Establecer los parámetros en una variable de entorno directamente en el shell que inicia la JVM.
- Estableciendo los parámetros en el script de inicio del servidor, por ejemplo, en $ WILDFLY_HOME / bin / standalone.sh o domain.sh.
- Estableciendo los parámetros en el archivo de configuración del servidor, es decir, en $ WILDFLY_HOME / bin / standalone.conf o domain.conf.

La primera alternativa en la práctica no es muy útil para cualquier otra prueba de nuevas variables.

Configuración de la JVM (cont)

La segunda alternativa puede ser utilizada, pero los parámetros previamente establecidos pueden perderse. Además, poner la configuración en la línea incorrecta puede estropear fácilmente el resto de la lógica en el script.

La última alternativa es la recomendada para prácticamente todas las situaciones, ya que la configuración se conserva y se versiona de una manera que separa la configuración de la lógica.

Es costumbre usar la variable de entorno, JAVA_OPTS, para establecer los parámetros de JVM. Todas las alternativas mencionadas hacen uso de esta variable, ya que es recogido y utilizado por las secuencias de comandos de inicio de WildFly.

Configuración de la JVM (cont)

Para no perder ningún parámetro VM previamente establecido, una buena práctica es configurar la variable de entorno para que se refiera a sí misma al agregar nuevos parámetros. Esto se hace usando la siguiente sintaxis:

JAVA_OPTS = "$ JAVA_OPTS <new-vm-params>"

El siguiente comando, como ejemplo, iniciará WildFly con un montón máximo de 2 GB, donde se ignorarán los parámetros de VM anteriores configurados previamente en JAVA_OPTS (se eliminan porque el nuevo valor establece la variable):

JAVA_OPTS = -Xmx2g

The thread pool executor subsystem

The thread pool executor subsystem

El subsistema el *pool thread executor* del ejecutor del grupo de subprocesos se introdujo en JBoss AS 7. Otros subsistemas pueden hacer referencia a los grupos de subprocesos configurados en este. Esto permite normalizar y administrar los grupos de subprocesos a través de los mecanismos de administración nativos de WildFly, y le permite compartir grupos de subprocesos entre subsistemas.

The thread pool executor subsystem :: tipos

Los siguientes grupos de subprocesos están disponibles:

- unbounded-queue-thread-pool
- bounded-queue-thread-pool
- blocking-bounded-queue-thread-pool
- queueless-thread-pool
- blocking-queueless-thread-pool
- scheduled-thread-pool

unbounded-queue-thread-pool

El *unbounded-queue-thread-pool thread pool executor* tiene el tamaño máximo y una cola ilimitada. Si el número de subprocesos en ejecución es menor que el tamaño máximo cuando se envía una tarea, se creará un nuevo subproceso. De lo contrario, la tarea se coloca en una cola. Esta cola puede crecer infinitamente.

max-threads : Máximo de hilos permitidos ejecutándose simultáneamente

keepalive-time : Especifica la cantidad de tiempo que los subprocesos del grupo deben mantenerse en funcionamiento cuando está inactivo. (Si no se especifica, los subprocesos se ejecutarán hasta que se cierre el ejecutor).

thread-factory : This specifies the thread factory to use to create worker threads.

bounded-queue-thread-pool

El ejecutor bounded-queue-thread-pool tiene un núcleo, un tamaño máximo y una longitud de cola especificada. Si el número de subprocesos en ejecución es menor que el tamaño del núcleo cuando se envía una tarea, se creará un nuevo subproceso; de lo contrario, se colocará en la cola. Si se ha alcanzado el tamaño máximo de la cola y no se ha alcanzado el número máximo de subprocesos, también se crea un nuevo subproceso. Si se golpea max-threads, la llamada se enviará al handoff-executor. Si no se configura handoff-executor, la llamada será descartada.

queue-length: Especifica el tamaño máximo de la cola.

max-threads: Especifica la cantidad máxima de hilos que pueden ejecutarse simultáneamente.

bounded-queue-thread-pool (cont)

queue-length: Especifica el tamaño máximo de la cola.

max-threads: Especifica el número máximo de subprocesos que pueden ejecutarse simultáneamente.

keepalive-time: Especifica la cantidad de tiempo que los subprocesos del grupo deben mantenerse en funcionamiento cuando está inactivo. (Si no se especifica, los subprocesos se ejecutarán hasta que se cierre el ejecutor).

handoff-executor: Especifica un ejecutor al que se delegarán las tareas, en el caso de que una tarea no pueda ser aceptada.

bounded-queue-thread-pool (cont)

allow-core-timeout : Especifica si los hilos centrales pueden agotar el tiempo de espera; si es falso, solo los subprocesos que estén por encima del tamaño del núcleo se agotarán.

thread-factory: Especifica la fábrica de subprocesos que se usará para crear subprocesos de trabajo.

blocking-bounded-queue-thread-pool

El ejecutor queuing-bounded-blocking tiene un núcleo, un tamaño máximo y una longitud de cola especificada. Si el número de subprocesos en ejecución es menor que el tamaño del núcleo cuando se envía una tarea, se creará un nuevo subproceso. De lo contrario, se pondrá en la cola. Si se ha alcanzado el tamaño máximo de la cola, se crea un nuevo subproceso; si no, se excede max-threads. Si es así, la llamada está bloqueada.

queueless-thread-pool

El queueless-thread-pool es un ejecutor de grupo de subprocesos sin ninguna cola. Si el número de subprocesos en ejecución es menor que max-threads cuando se envía una tarea, se creará un nuevo subproceso; de lo contrario, se llamará al ejecutor de transferencia. Si no se configura ningún handoff-executor, la llamada será descartada.

keepalive-time: Especifica la cantidad de tiempo que los subprocesos del grupo deben mantenerse en funcionamiento cuando está inactivo. (Si no se especifica, los subprocesos se ejecutarán hasta que se cierre el ejecutor).

thread-factory: Especifica la fábrica de subprocesos que se usará para crear subprocesos de trabajo

handoff-executor: Especifica un ejecutor al que se delegarán las tareas, en el caso de que una tarea no pueda ser aceptada.

blocking-queueless-thread-pool

El ejecutor blocking-queueless-thread-pool no tiene ninguna cola. Si el número de subprocesos en ejecución es menor que max-threads cuando se envía una tarea, se creará un nuevo subproceso. De lo contrario, la persona que llama será bloqueada.

keepalive-time: Especifica la cantidad de tiempo que los subprocesos del grupo deben mantenerse en funcionamiento cuando está inactivo. (Si no se especifica, los subprocesos se ejecutarán hasta que se cierre el ejecutor).

thread-factory: Especifica la fábrica de subprocesos que se usará para crear subprocesos de trabajo

scheduled-thread-pool

El ejecutor scheduled-thread-pool es utilizado por tareas que están programadas para activarse en un momento determinado..

keepalive-time: Especifica la cantidad de tiempo que los subprocesos del grupo deben mantenerse en funcionamiento cuando está inactivo. (Si no se especifica, los subprocesos se ejecutarán hasta que se cierre el ejecutor).

thread-factory: Especifica la fábrica de subprocesos que se usará para crear subprocesos de trabajo

Tuning Undertow

Tuning Undertow

Wildfly maneja el tráfico en diferentes protocolos a través de los listeners. El único que está habilitado out of the box es HTTP. Si se necesita apoyo para AJP y HTTPS, tendrá que ser configurado y habilitado los listeners correspondientes. En versiones anteriores del servidor de aplicaciones, hubo algunos ajustes bastante que tenían que ver con la optimización del rendimiento de estas configuraciones de protocolos (principalmente ajustes de grupos de subprocesos). Este no es el caso en Undertow como el manejo hilo se maneja anteriormente en la pila por el subsistema de I / O y el uso de XNIO.

Worker

Componente es el worker. El valor real nombre del worker por defecto para un listener específico (en este ejemplo, el: http-listener) puede ser recuperada por el siguiente comando CLI:

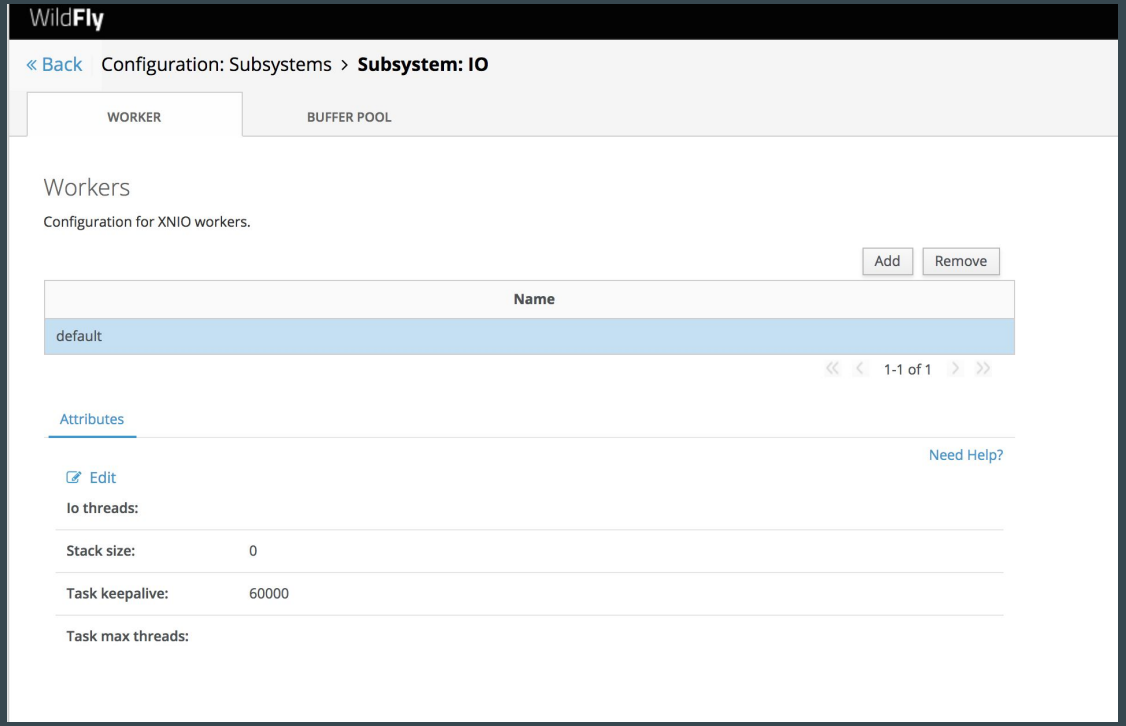
```
/subsystem=undertow/server=default-server/http-listener=default:read-attribu
te(name=worker)
{
  "outcome" => "success",
  "result" => "default"
}
```

Muchos workers se pueden definir en Wildfly, y cada worker puede servir para uno o más listeners.

Worker

La misma información también se puede explorar en la consola de administración

Ver documento



Servlet Container and JSP compilation

Cache Tuning

Logging tuning

EJB connection pool

Tuning the database connection pool

MBeans

Configuración de Wildfly mediante MBenas

Existe un sitio muy recomendable para saber qué configuración está disponible vía MBeans https://wildscribe.github.io/WildFly/12.0/index.html

Apache JMeter

Best Practices

Stress con JMeter

Stressing

Existe un gran problema con los planes de stress de máquinas virtuales y es que el mismo JVM del a JMeter o la cantidad de datos almacenados en memoria por el mismo GUI. Para ello, JMeter nos provee de una opción donde no inicia el GUI y ejecuta el plan creando un reporte.

```
================================================================================
Don't use GUI mode for load testing !, only for Test creation and Test debugging.
For load testing, use NON GUI Mode:
   jmeter -n -t [jmx file] -l [results file] -e -o [Path to web report folder]
& increase Java Heap to meet your test requirements:
   Modify current env variable HEAP="-Xms1g -Xmx1g -XX:MaxMetaspaceSize=256m" in the jmeter batch file
Check : https://jmeter.apache.org/usermanual/best-practices.html
================================================================================
```

APM

...

DataDog

Java Agent

https://docs.datadoghq.com/tracing/setup/java/

Tomcat

https://www.datadoghq.com/blog/monitor-tomcat-metrics/

New Relic

For Java

https://docs.newrelic.com/docs/agents/java-agent/getting-started/introduction-new-relic-java

App Dynamics

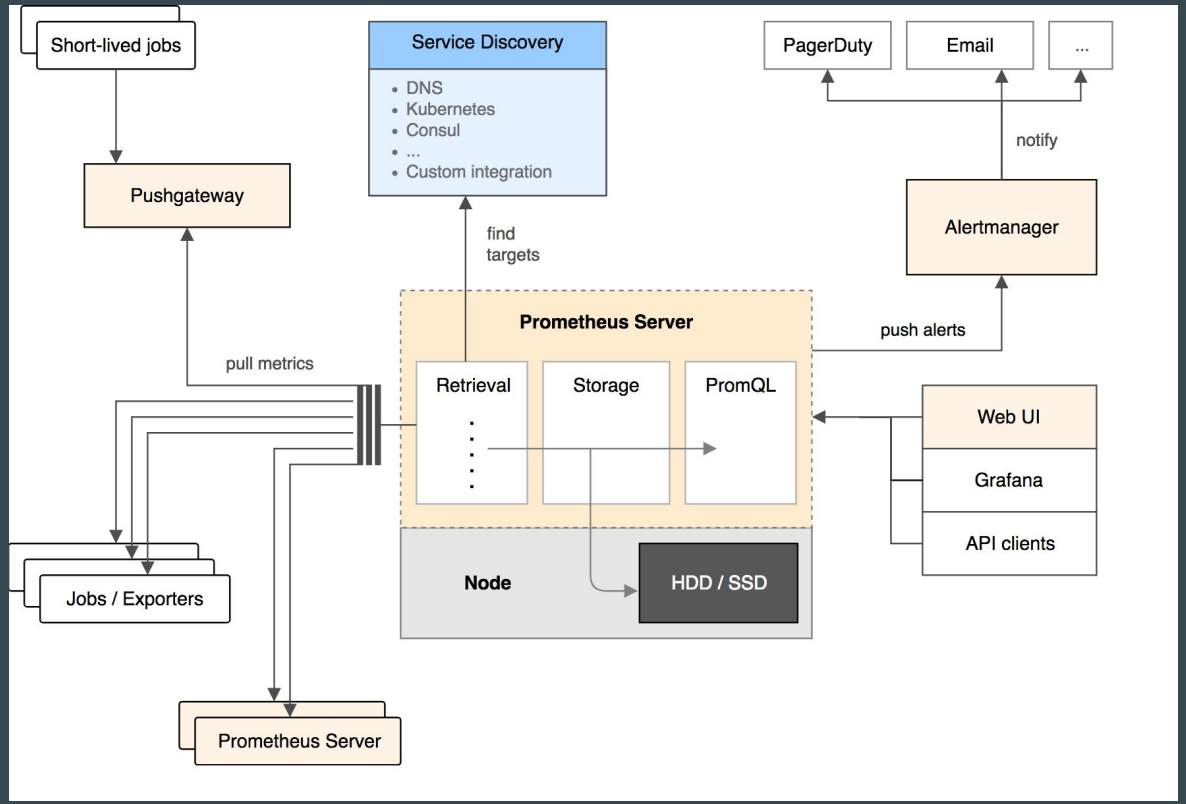
Links

https://www.appdynamics.com/

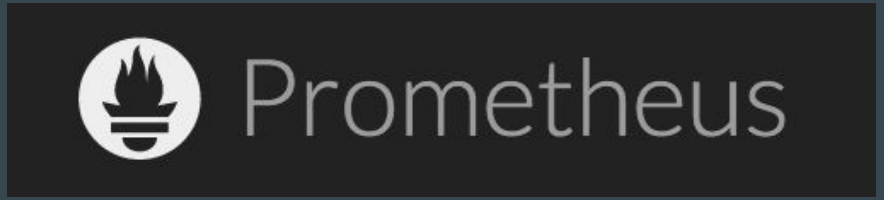
https://www.appdynamics.com/pricing/

Prometheus

Prometheus es un conjunto de herramientas de supervisión y alerta de sistemas de código abierto con un ecosistema activo.



Prometheus



Links

https://www.digitalocean.com/community/tutorials/how-to-install-prometheus-using-docker-on-ubuntu-14-04