Performance tuning EJBs en WildFly

La mayor parte del ajuste del rendimiento de los EJB en WildFly consta de varias pools de ajuste. En las siguientes secciones se irán más en detalles sobre cada uno de los tipos de EJB y su ajuste real. Primero, sin embargo, vamos a empezar con información genérica sobre la habilitación de estadísticas detalladas en WildFly y algunas optimizaciones de llamadas a métodos locales y remotos.

Permitir obtener estadísticas detalladas

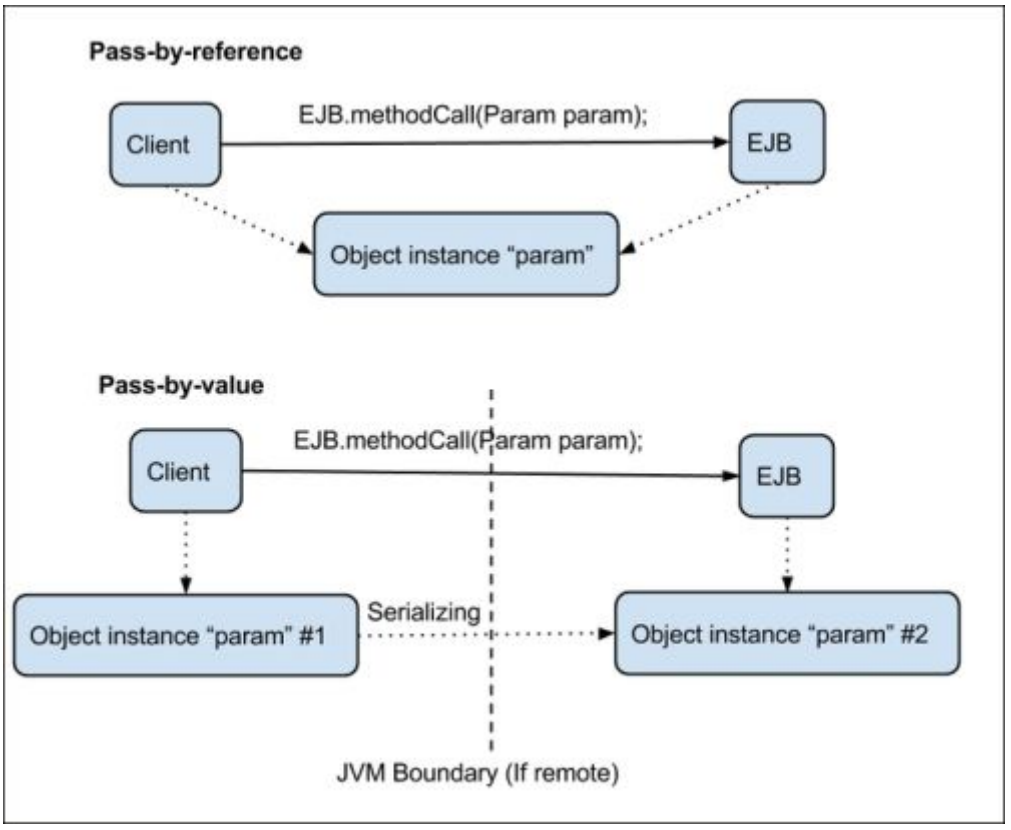
Por defecto, las estadísticas detalladas no se generan en WildFly. Para generar estadísticas, se debe activar con el siguiente comando CLI:

/subsystem=ejb3:write-attribute(name=enable-statistics,value=true)

Optimizaciones de llamadas Local y Remote

Los Session Beans tienen dos interfaces posibles, @Local y @Remoto. La diferencia es que cuando se utiliza la interfaz local, que hace posible que el servidor de aplicaciones para llevar la llamada a cabo un *pase por referencia* en lugar de una *pase por valor.* Al utilizar referencias, los parámetros del método de llamada se envían como referencias de memoria (como cualquier otra llamada estándar de Java dentro de la misma máquina virtual), pero cuando se utiliza la interfaz remota, todos los parámetros tienen que ser serializado.

El enfoque de paso-por-valor es fácil de entender cuando se utilizan los clientes remotos como referencias de memoria no funciona, pero el hecho es que si la interfaz de control remoto se usa en el servidor de aplicaciones tendrá que pasar por el proceso de serialización, como este es requerido por la especificación EJB3. Por lo tanto, la recomendación es que siempre se utiliza la interfaz local cuando sea posible. El diagrama siguiente ilustra la diferencia entre los dos enfoques:



pasar-por-referencia frente pasar-por-valor

En el caso *pasar-por-referencia*, el **cliente** hace una llamada de método local. No se necesita ninguna serialización. Sólo la referencia de (o dirección a) la instancia de objeto para el parámetro es en realidad pasan al EJB. El EJB entonces accede a la misma instancia de objeto situado como la posición en la memoria como el **cliente** utiliza y se hace referencia.

El uso de *pasar-por-valor*, el parámetro de instancia de objeto se serializa y copiado desde el **cliente** al **EJB,** donde se deserializa y se utiliza, como el **cliente** realiza la llamada de método a la **EJB.** Aquí, hay más de una instancia de objeto del parámetro en la memoria.

Como se mencionó anteriormente, los clientes remotos necesitan usar la interfaz remota, pero en realidad, todos los clientes que se encuentran fuera del archivo EAR están no pueden de usar la interfaz local, incluso si se están ejecutando en el mismo servidor de aplicaciones.

Desde un punto de vista tecnológico, es posible que un servidor de aplicaciones, para optimizar las llamadas de interfaz remotas dentro de la misma VM y utilizar *pasar-por-referencia* en lugar de *pasar-por-valor*; sin embargo esto rompe el contrato especificación EJB. Como tal, sólo se debe utilizar después de un examen a fondo de la arquitectura y los planes futuros de la plataforma (como las versiones actualizadas del servidor de aplicaciones podrían no incluir esta funcionalidad).

Para permitir la optimización de JBoss/Wildfly, puede ejecutar el siguiente comando CLI:

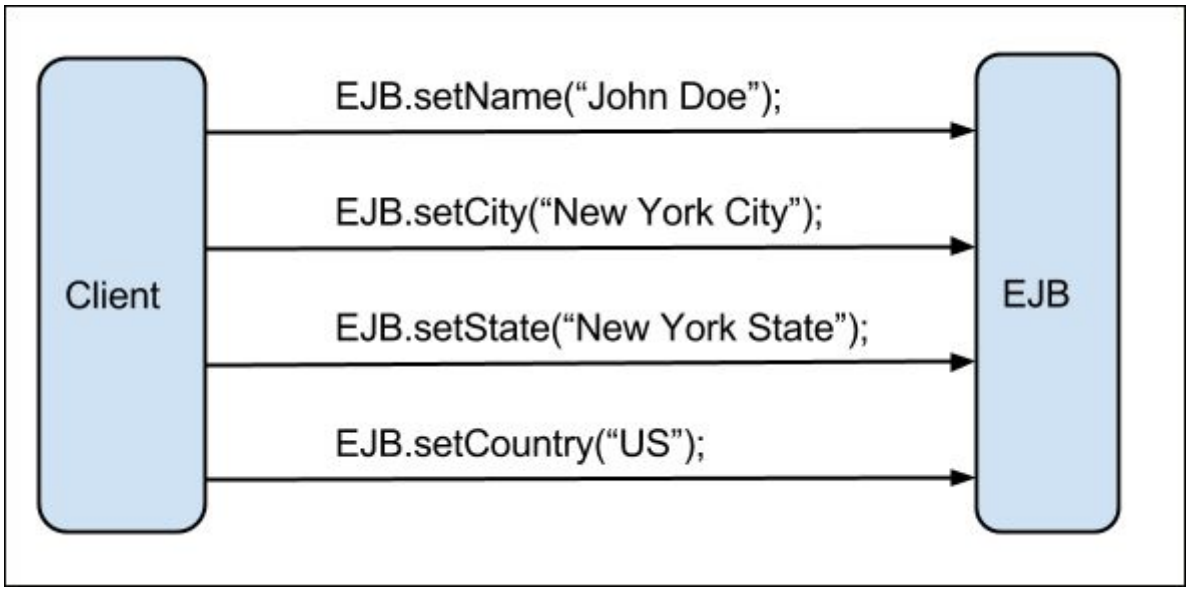
/subsystem=ejb3:write-attribute(name=in-vm-remote-interface-invocation-pass-by-value, value=false)

Nota

Tenga en cuenta que *pasar-por-referencia* sólo será utilizada si el cliente y EJB tienen acceso a las mismas definiciones de clase (es decir clases idénticas cargadas desde el mismo cargador de clases). El contenedor realizará una comprobación de poca profundidad para verificar si la optimización se activa y volverá a la llamada no optimizado si es necesario. Una revisión superficial, en contraste con un cheque de profundidad, sólo comprueba el objeto de nivel superior y no seguir comprobando las referencias dentro de ella a los demás. Esto significa que usted puede obtener *ClassCastException*, por lo que sólo lo utilizan si está seguro de que el cliente está utilizando el mismo cargador de clases como EJB.

Si se utilizan las interfaces remotas, es una buena idea para reducir al mínimo el número de llamadas de red realizados utilizando el *enfoque de bean grueso*.

Mirando el diagrama siguiente, vemos un ejemplo de un cliente que realiza muchas llamadas a un EJB, el establecimiento de un atributo a la vez:

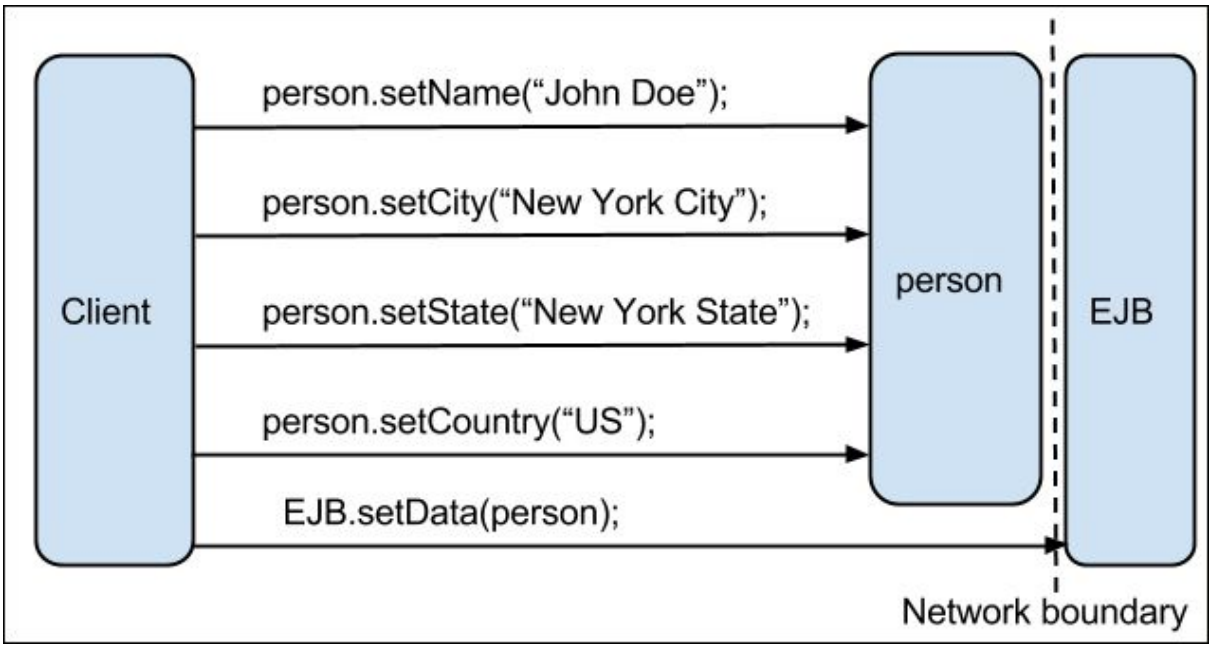


Enfoque de bean fino

El diagrama anterior es un ejemplo de lo que se considera que es un *enfoque de bean fino*. Esto funciona relativamente bien si las interfaces locales (y *pasar por referencia*) se utilizan, ya que habrá local en-VM llamadas a métodos. Sin embargo, si el acceso de los clientes a un EJB a través de interfaces remotas (y usando el paso por valor), las llamadas irá través de la red, lo que resulta en costos relativamente altos relacionados con el rendimiento.

El enfoque de bean grueso se ilustra en el diagrama siguiente, se utiliza un objeto de datos (persona) que se rellena con información sobre el lado del cliente (utilizando todos

despliegue local y en-VM llama) y es enviado luego a la EJB en una sola llamada remota (EJB.setData (persona)). Esto reduce al mínimo las llamadas relacionadas con las redes necesarias y mejora el rendimiento y la eficiencia general.



Enfoque de bean grueso

Session beans y transacciones

Si está utilizando **transacción gestionada por contenedor (CMT),** que es la estrategia de transacción por defecto aproximación, el contenedor se iniciará automáticamente una transacción para usted porque el atributo de transacción por defecto es necesario. Esto garantiza que el trabajo realizado por el método está dentro de un contexto de transacción global.

Sin embargo, la gestión de transacciones es un asunto costoso y hay que verificar si sus métodos EJB realmente necesitan una transacción. Por ejemplo, un método que simplemente devuelve una lista de objetos para el cliente por lo general no necesita ser alistado en una transacción.

Por esta razón, se considera una buena práctica en el ajuste para eliminar las transacciones innecesarios de su EJB. Por desgracia, esto es a menudo subestimada por los desarrolladores que les resulta más fácil definir una política de transacción genérica para todos los métodos en el descriptor de despliegue. Por ejemplo, en el siguiente ejemplo de configuración, todos los métodos de TestEJB utilizarán el atributo de transacción requerida:

```
<container-transaction>
  <method>
    <ejb-name>TestEJB</ejb-name>
    <method-name>*</method-name>
```

```
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
```

Sin embargo, con el uso de anotaciones EJB3, ya no hay excusa para su negligencia; es posible deshabilitar el valor por defecto mediante ell tipo de transacción a NOT_SUPPORTED con una simple anotación, como se muestra en el siguiente código:

```
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public List<PayRoll> findAll()
{...}
```

Los problemas comunes con las aplicaciones por lotes de larga duración que incluyen la manipulación de transacciones. En WildFly estas pueden verse afectadas por el *transaction timeout* predeterminada de 300 segundos (5 minutos). En lugar de simplemente aumentar esta configuración que afecta a todas las transacciones en el servidor de aplicaciones, es mejor para anular el tiempo de espera para un EJB en particular que inicia la transacción. Una llamada "colgada" puede bloquear recursos por un largo tiempo, recursos que son necesarios por otras llamadas, lo que ralentiza la aplicación. Mantener el tiempo de espera de transacción a un nivel razonable para las llamadas estándar y el aumento de la pena sólo por las específicas; esto minimiza el impacto de este suceso. En cambio, los recursos bloqueados se dará a conocer a una temprana de tiempo de espera.

El tiempo de espera predeterminado puede ser gestionado por el siguiente comando CLI:

```
/subsystem=transactions:read-attribute(name=default-timeout)
{
   "outcome" => "success",
   "result" => 300
}
```

La configuración del timeout a una valor específico se puede hacer mediante el siguiente comando CLI (estamos poniéndolo a 600 segundos aquí):

```
/subsystem=transactions:write-attribute(name=default-timeout, value=600)
```

Un valor de tiempo de espera específico puede también ser configurado de código con la siguiente anotación:

```
@TransactionTimeout(value = 600, unit = TimeUnit.SECONDS)
```

O, se puede configurar en el archivo de configuración jboss-ejb3.xml utilizando el código siguiente:

```
<container-transaction>
  <method>
    <ejb-name>MyBean</ejb-name>
    <method-name>*</method-name>
  </method>
  <tx:trans-timeout>
    <tx:timeout>600</tx:timeout>
    <tx:unit>Seconds</tx:unit>
  </tx:trans-timeout>
</container-transaction>
...
```

Nota

Tenga en cuenta que sólo es válido cuando se utiliza la transacción atributos requeridos (si el *bean* actual es el verdadero creador de la transacción) o REQUIRES_NEW (si es una nueva transacción se debe crear para cada llamada del *bean*).

En JBoss/Wildfly, es posible activar las estadísticas de las transacciones. Se está desactivado por defecto por razones de rendimiento, pero se puede activar mediante el siguiente comando CLI y luego reiniciar el servidor de aplicaciones:

/subsystem=transactions:write-attribute(name=enable-statistics, value=true)

Ejecutar el siguiente comando CLI a la lista atributos las estadísticas de transacciones disponibles:

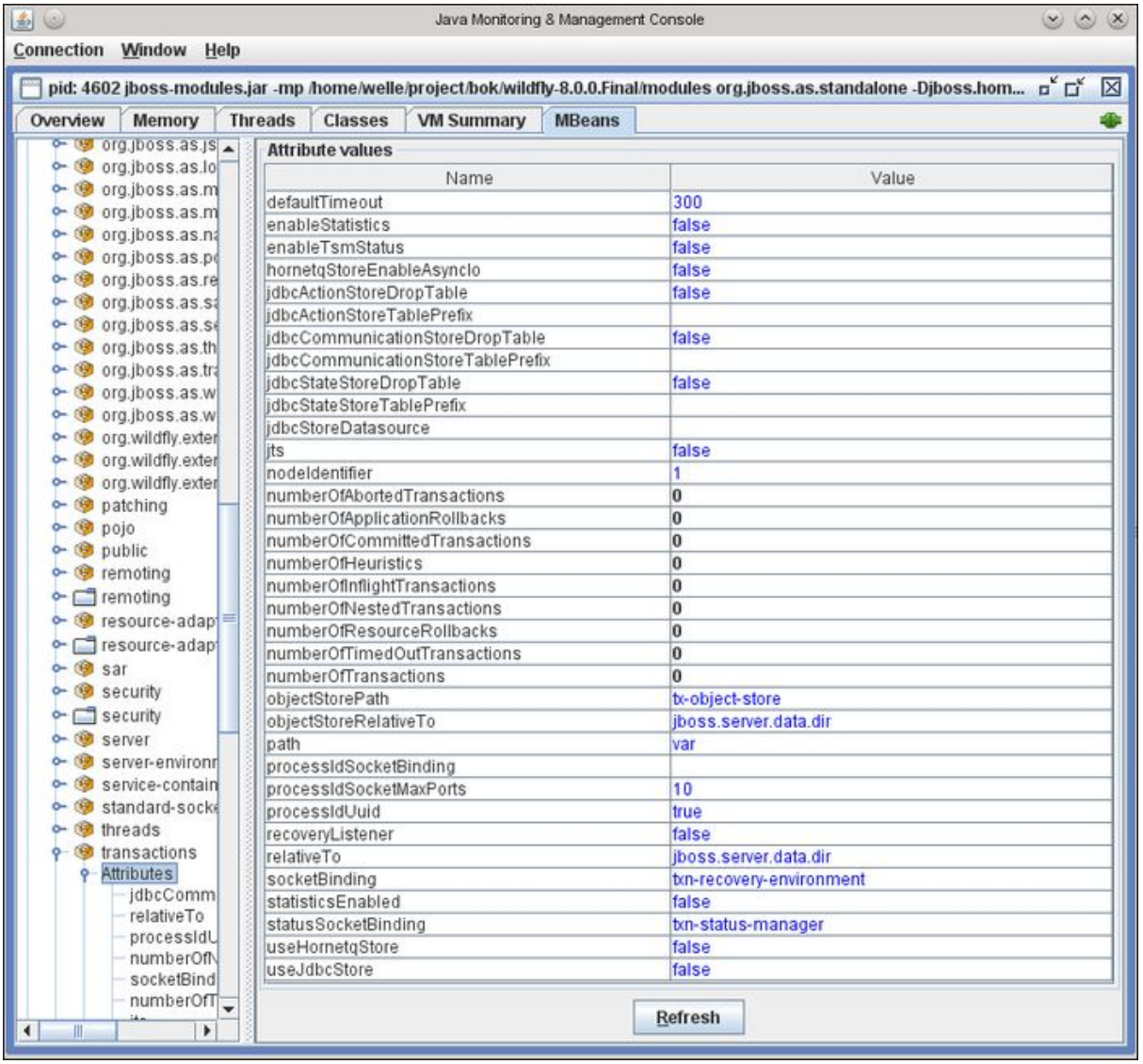
/subsystem=transactions:read-resource-description

los atributos que pueden ser de interés para el seguimiento son los siguientes:

- number-of-transactions
- default-timeout
- number-of-application-rollbacks
- number-of-aborted-transactions
- number-of-inflight-transactions
- number-of-timed-out-transactions
- number-of-committed-transactions

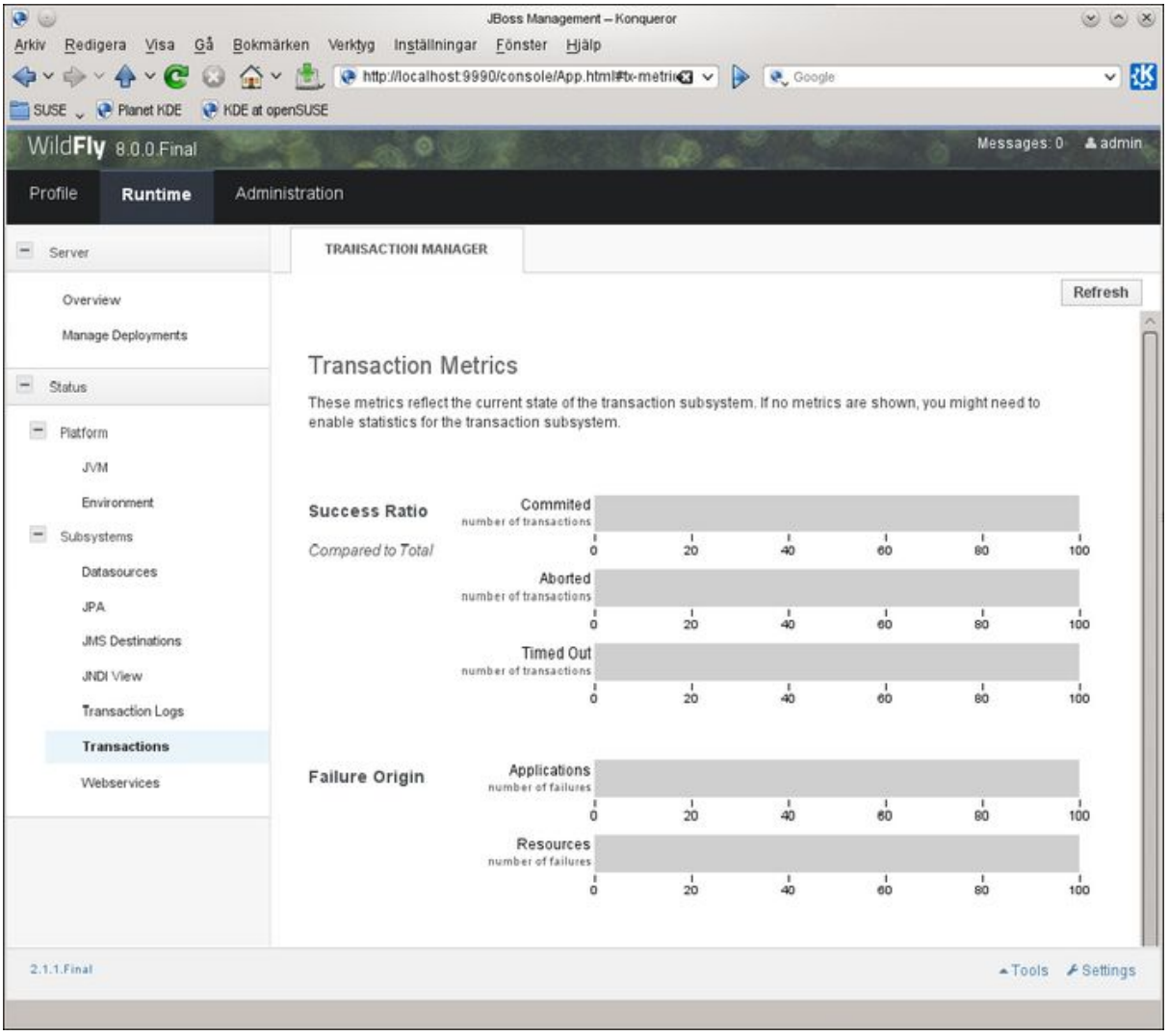
- number-of-resource-rollbacks

a MBean JMX con los jboss.as:subsystem=transactions ObjectName también está disponible para el monitoreo . Un ejemplo utilizando JConsole para el listado y sus atributos se muestra en la siguiente captura de pantalla:



MBean de información de transacción

en la siguiente captura de pantalla, las **métricas de transacción** ver en **Management Console** proporciona una visión global de las transacciones:



Información sobre las transacciones en Consola de administración de

llamadas EJB remotos

en JBoss/Wildfly , hay un grupo de subprocesos especial que se utiliza para todos los accesos remotos que necesitamos para sintonizar. Este pool consta de un tamaño max-hilos y una cola (de tareas) sin límite superior. Si no hay temas están disponibles para una tarea, esta tarea se pone en la cola sin ningún tiempo de espera. Por defecto, este grupo de subprocesos se establece en un máximo de 10 hilos simultáneos, lo cual es un valor que es a menudo demasiado bajo para una configuración de producción.

Los dos siguientes comandos CLI se puede utilizar para leer y cambiar la configuración max-hilos:

 # Get max threads
/subsystem=ejb3/thread-pool=default:read-attribute(name=max-treads)

```
{
   "outcome" => "success",
   "result" => 10
}
# Set max threads
/subsystem=ejb3/thread-pool=default:write-attribute(name=max-threads, value=100)
```

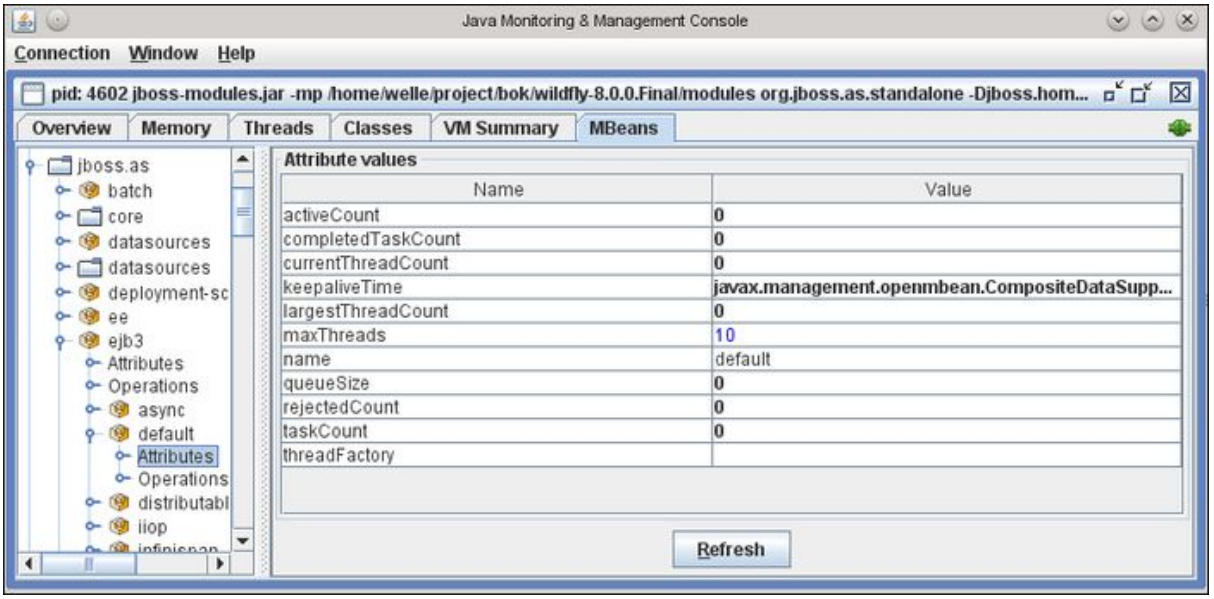
Este grupo de subprocesos también se puede controlar mediante la CLI. Ejecute el siguiente comando CLI para listar los atributos disponibles para la supervisión:

```
/subsystem=ejb3/thread-pool=default:read-resource-description
```

Los atributos que pueden ser de interés para el seguimiento son los siguientes:

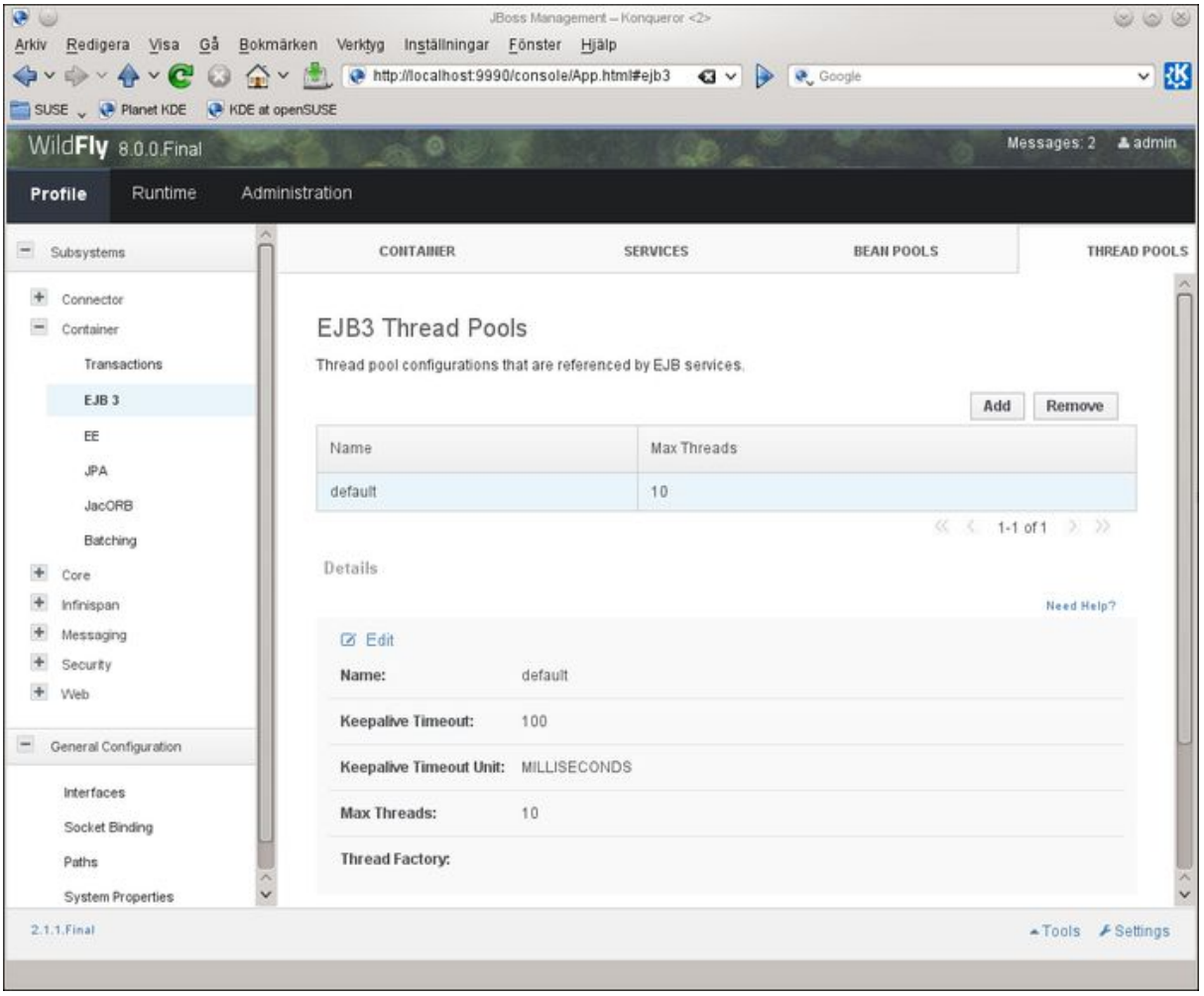
- max-threads
- largest-thread-count
- current-thread-count
- activeCount
- task-count
- completed-task-count
- rejected-count
- queue-size

Un JMX MBean con el objectname jboss.as:subsystem=ejb3,thread-pool=default también está disponible para el monitoreo. La siguiente captura de pantalla muestra cómo se utiliza JConsole:



El grupo de subprocesos EJB3 MBean

la siguiente captura de pantalla que muestra **la Consola de** administración,que sólo proporciona una vista de configuración en este pool y no cualquier información de tiempo de ejecución:



El grupo de subprocesos EJB3 en Management Console

Como este grupo de subprocesos se utiliza para asíncrono EJB3 y las llamadas de temporizador, así como para el acceso remoto, puede que tenga que investigar, incluso si usted no tiene ningún clientes remotos en su aplicación.

Para el resto de este capítulo, los resultados dependen del parámetro max-hilos en el entorno del pool hilo EJB3 que se establece en 100. Si no se establece en 100, ninguna de las pruebas será capaz de ejecutar más de 10 llamadas simultáneas.

La optimización de Stateless Session Beans

Si SLSB no es costoso para crear una instancia (sin @PostConstruct pesada), la puesta en común puede ser más lenta que la creación de nuevas instancias cuando sea necesario. Este comportamiento es en realidad defecto en JBoss/Wildfly, y significa que cada hilo llamante crea su propia instancia de la SLSB necesario.

Las desventajas son que puede ser contraproducente cuando SLSB tarda mucho tiempo en ser creado o en un escenario en el que tiene recursos que están muriendo de hambre. Por

ejemplo, puede que tenga que tener un control exacto de su SLSB si son estrictamente dependiente de un recurso externo, como una cola JMS.

Cómo activar el grupo de instancia para SLSB se puede hacer mediante el siguiente comando CLI que activa un pool ya enviadas y configurado:

/subsystem=ejb3:write-attribute(name=default-slsb-instance-pool,value=slsb-strict-max-pool)

el enfoque que se utilizará depende de una gran cantidad de criterios y la mejor decisión se basa (como de costumbre) en pruebas reales. Como no hay mucho para optimizar si se utiliza el comportamiento predeterminado, además de minimizar las devoluciones de llamada de ciclo de vida, nos centramos el resto de la discusión sobre un escenario cuando se habilita SLSB puesta en común.

Al habilitar SLSB puesta en común, el máximo tamaño del pool defecto en JBoss/Wildfly se establece en 20.

Los siguientes dos comandos CLI recuperar y establecer el tamaño máximo del grupo:

 # Get the max pool size
/subsystem=ejb3/strict-max-bean-instance-pool=slsb-strict-max-pool:read-attribute(name=max-pool-size)
 # Set the max pool size
/subsystem=ejb3/strict-max-bean-instance-pool=slsb-strict-max-pool:write-attribute(name=max-pool-size, value=30)

Si no hay una instancia disponible en el pool, la persona que llama se pone en espera para un máximo de 5 minutos (por defecto). Esto se puede cambiar mediante la ejecución de los siguientes comandos de la CLI:

 # Get the unit for the timeout value
/subsystem=ejb3/strict-max-bean-instance-pool=slsb-strict-max-pool:read-attribute(name=timeout-unit)
 # Get the timeout value
/subsystem=ejb3/strict-max-bean-instance-pool=slsb-strict-max-pool:read-attribute(name=timeout)
 # Set the timeout value
/subsystem=ejb3/strict-max-bean-instance-pool=slsb-strict-max-pool:write-attribute(name=timeout, value=10L)

En los siguientes ejemplos, que le mostrará cómo un pool sLSB se puede controlar. SLSB se nombra *StatelessSessionTestBean* y se encuentra en el ejemplo artifact MyEJB.jar, que a su vez, se despliega dentro EAR, MyEAR.ear.

Utilice el siguiente comando CLI para explorar los atributos disponibles del pool SLSB para cualquier bean desplegado:

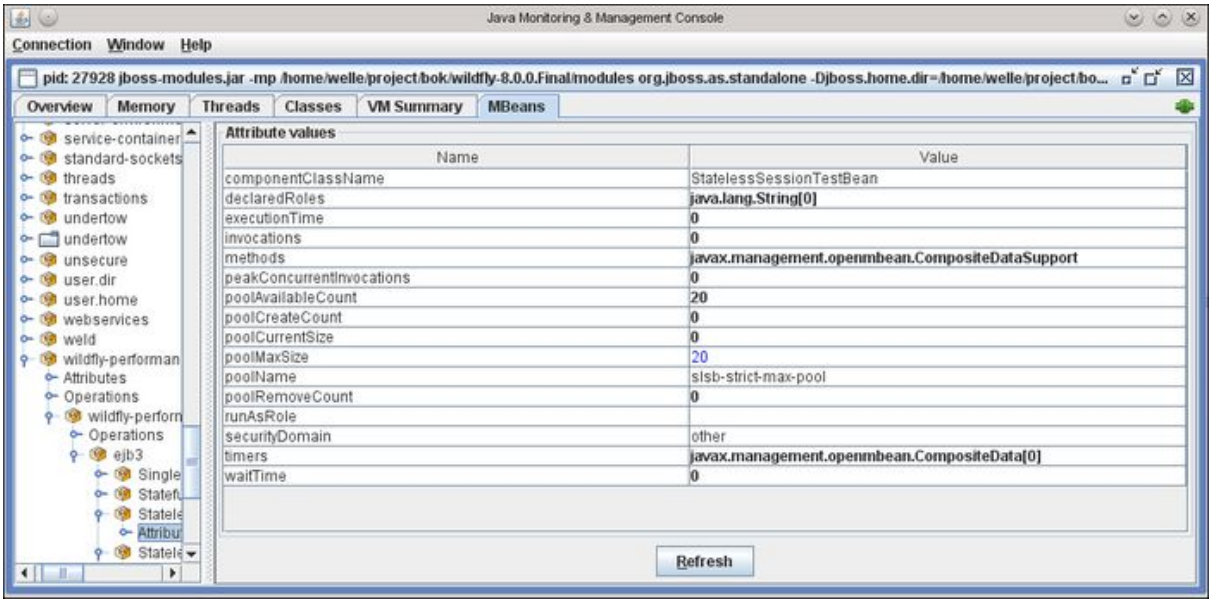
/deployment=MyEAR.ear/subdeployment=MyEJB.jar/subsystem=ejb3/stateless-session-bean=StatelessSessionTestBean:read-resource-description

los atributos que pueden ser de interés para el seguimiento son los siguientes:

- pool-max-size
- pool-current-size
- pool-available-count
- peak-concurrent-invocations
- invocations
- pool-create-count
- pool-remove-count

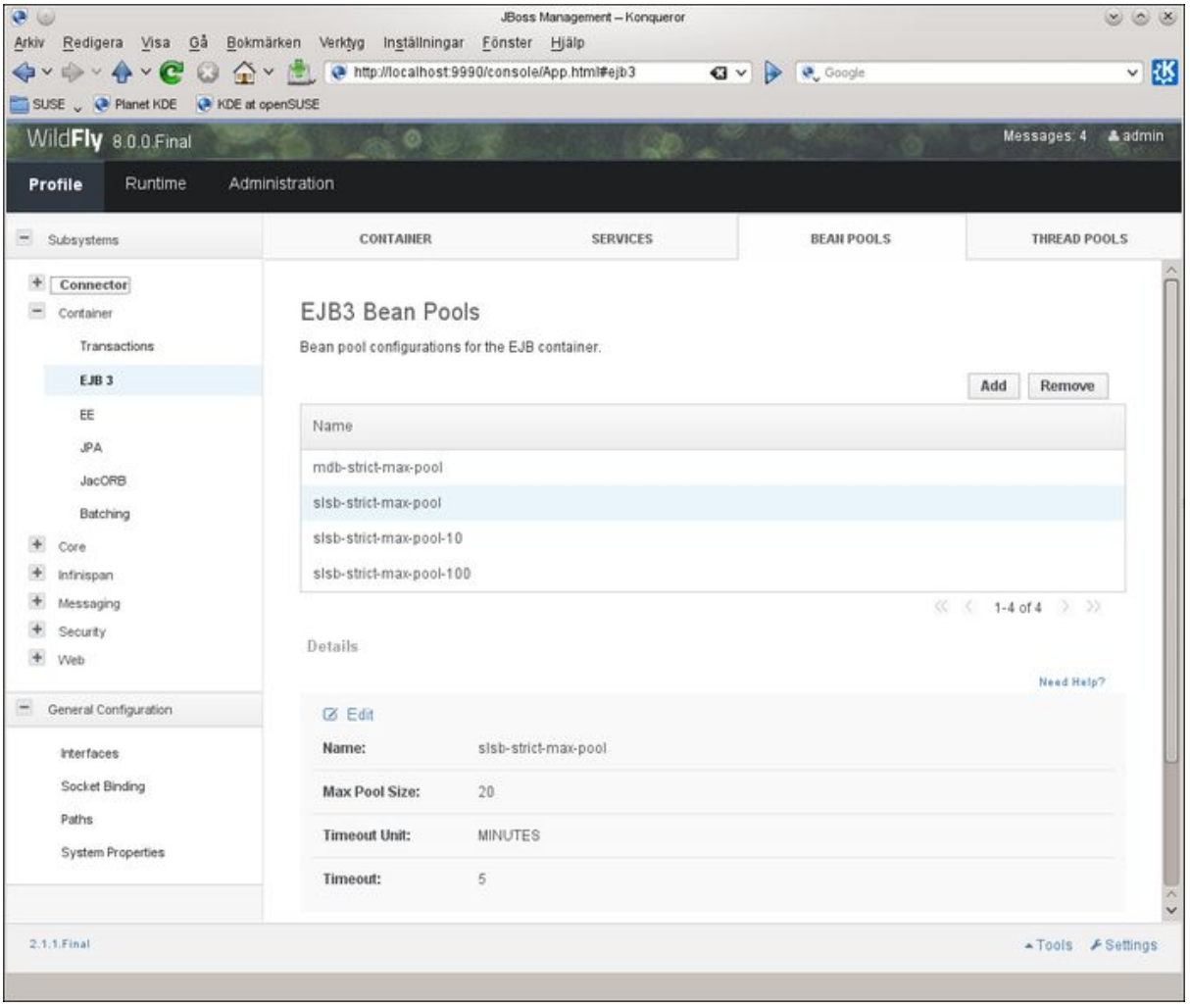
para encontrar el valor de la carga máxima de un SLSB ya desplegada, echar un vistazo al atributo pico concurrente-invocaciones.

Como es habitual, también hay un MBean disponibles para proporcionar la misma información. En este caso el JMX MBean tiene la ObjectName jboss.as:deployment=MyEAR.ear,subdeployment=MyEJB.jar,subsystem=ejb3,stateless-session-bean=StatelessSessionTestBean, y su salida se puede ver utilizando JConsole como se muestra en la siguiente captura de pantalla:



Estadística MBean para un SLSB

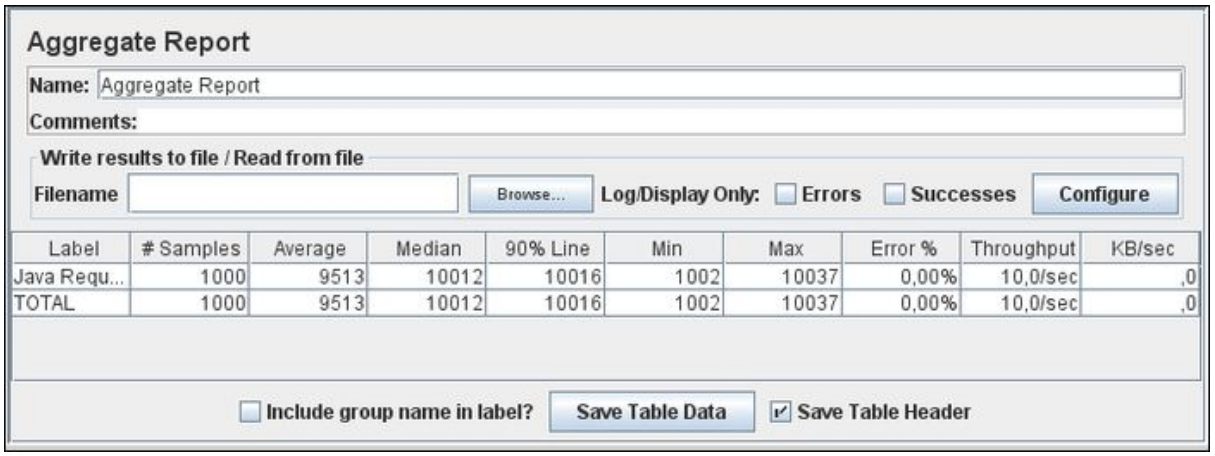
por desgracia, la **consola de administración** sólo proporciona una vista de configuración en este pool y no cualquier información de tiempo de ejecución. Las posibilidades de configuración se pueden ver en la siguiente captura de pantalla:



vista de la Consola de administración de un pool SLSB

Tuning el Pool SLSB

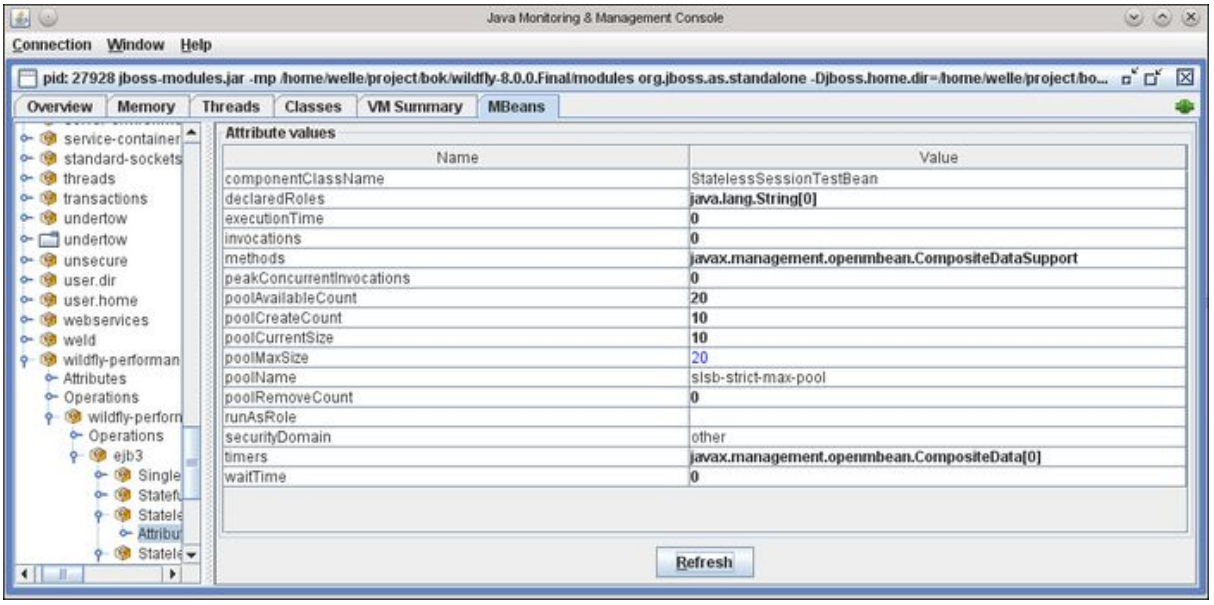
vamos a ejecutar un caso de prueba con 100 hilos simultáneos de llamar a un SLSB que simplemente dormir durante 1 segundo, donde cada hilo ejecuta 10 iteraciones sin ninguna pausa entre. La siguiente captura de pantalla que muestra el informe de JMeter de esta prueba:



El resultado SLSB JMeter

Para los ejemplos de este capítulo, nos centraremos en la de **rendimiento.** columna En la vida real, es probable que desee para investigar las otras medidas también. Por ejemplo, si los valores de tiempo de llamada(media, mediana, **90%**Line, **Min,**y **Max)**se puede comprobar con el fin de encontrar los valores extremos que pueden indicar alguna agotamiento de recursos, problemas de sincronización, y así sucesivamente.

Así que vamos a pasar a la de **rendimiento.** columna No debe ser el caudal de alrededor de 20 invocaciones / seg como el tamaño del pool es de 20? Veamos el uso del pool para el uso de la SLSB JMX MBean durante la prueba utilizando JConsole, como se muestra en la siguiente captura de pantalla:



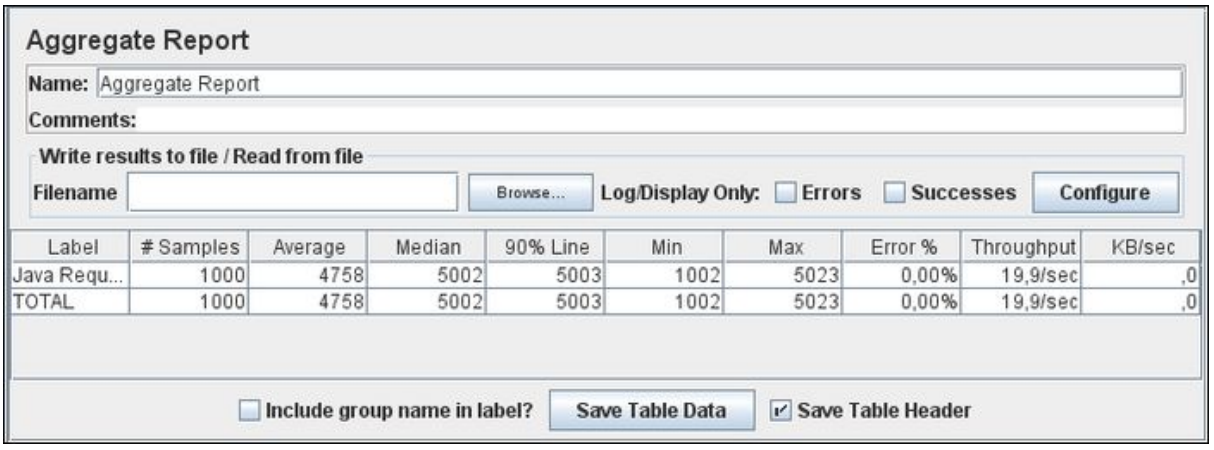
MBean para SLSB en la prueba de

JBoss/Wildfly sólo ha creado 10 casos en el pool, pero no debería ser 20?

La respuesta es que en realidad alcanzado el límite de la agrupación de hebras remota EJB se mencionó al principio de este capítulo. Vamos a cambiarlo a 100 usando el comando CLI:

/subsystem=ejb3/strict-max-bean-instance-pool=slsb-strict-max-pool-100:add(max-pool-size=100, timeout-unit=MINUTES, timeout=5L)

Después del cambio y volver a la prueba, la siguiente captura de pantalla muestra el resultado de JMeter:



el resultado de la JMeter SLSB prueba con el aumento del pool EJB3

Ahora, hemos llegado al rendimiento esperado de alrededor de 20 invocaciones / seg, y viendo la JMX MBean para el SLSB confirma que 20 casos se han creado y utilizado.

Entonces, ¿qué si el tamaño predeterminado de 20 no es suficiente para la aplicación. Una solución sería aumentar el valor predeterminado. Esto, sin embargo, afectar a todos SLSB en el contenedor. También es posible crear un nuevo grupo y dejar que el uso individual que SLSB pool en su lugar. En primer lugar, se crea una nueva agrupación denominada slsb-estricta-max-pool-100, con el siguiente comando CLI:

/subsystem=ejb3/strict-max-bean-instance-pool=slsb-strict-max-pool-100:add(max-pool-size=100, timeout-unit=MINUTES, timeout=5L)

Entonces, podemos utilizarlo en nuestro bean con la anotación específica JBoss/Wildfly / JBoss/Wildfly, como se muestra en el siguiente código:

```
@org.jboss.ejb3.annotation.PPool(value="slsb-strict-max-pool-100")
```

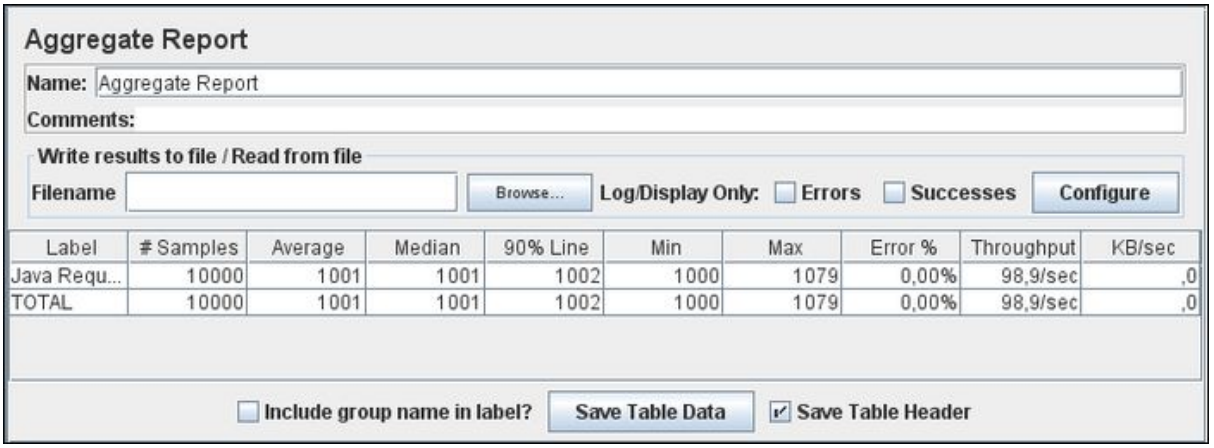
O bien, podemos utilizar el archivo de configuración jboss- ejb3.xml como se muestra en el siguiente código:

```
<p:pool>
  <ejb-name>MyBean</ejb-name>

<p:bean-instance-pool-ref>slsb-strict-max-pool-100</p:bean-instance-pool-r
ef>
</p:pool>

...
```

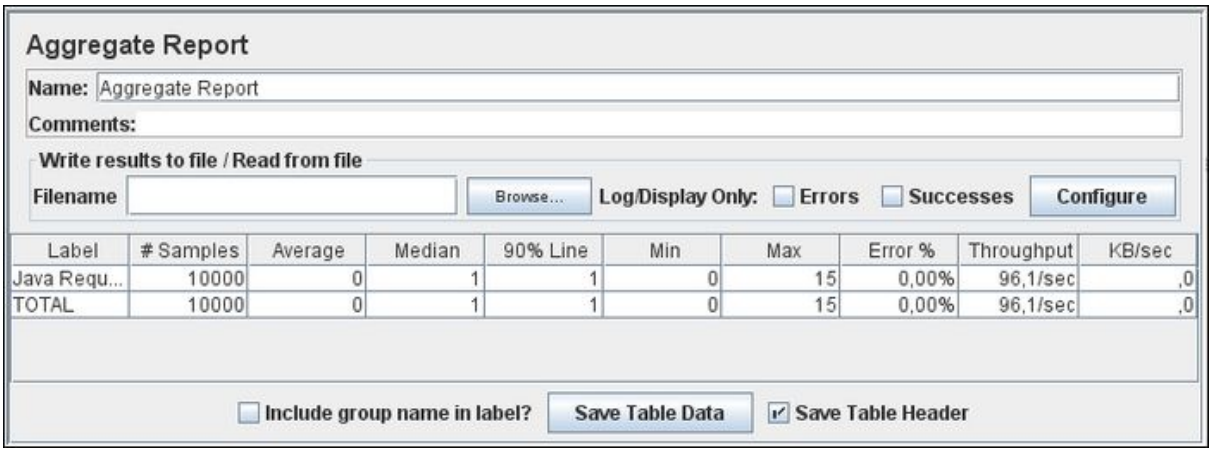
Ahora que realmente son conseguir este bean para volar. Una nueva prueba nos muestra los resultados de JMeter en la siguiente captura de pantalla:



Los resultados de JMeter del SLSB prueba con el aumento de la agrupación de instancias

Con todo esto dicho, no hay que olvidar que estos casos de prueba son un poco irreal. Clientes muy rara vez llaman a la misma SLSB una y otra vez sin pausa. Por lo tanto, normalmente no hay necesidad de correlacionar ciegamente el tamaño del pool con el número esperado de personas que llaman.

Vamos a ejecutar un caso de prueba con 100 hilos simultáneos, que llama SLSB con pool instancia establece en 10. Cada persona que llama hilo duerme durante un segundo entre llamadas y la SLSB devolver la ejecución inmediata sin dormir. El resultado de JMeter se muestra en la siguiente captura de pantalla:



Los resultados de una prueba de JMeter SLSB más realista

de la **rendimiento** columna que muestra que un grupo de 10 es probablemente suficiente para este escenario, ya que coincida con la carga que debe generar 100 llamadas / seg.

Optimización de los Stateful Session Beans

Como hemos comentado anteriormente, la SFSB el proceso de *passivation* requiere gastos generales y constituye un impacto en el rendimiento de la aplicación. Si el SFSB pasivado posteriormente se requiera la aplicación, el contenedor se activa mediante la restauración desde el disco.

Al eliminar explícitamente SFSB cuando haya terminado, aplicaciones disminuirán la necesidad de la *passivation*, minimizar los gastos de contenedores, y mejorar su rendimiento. Además, mediante la eliminación de forma explícita SFSB, no es necesario depender de los valores de tiempo de espera para la eliminación de los beans rancios.

La configuración por defecto de *passivation* en JBoss/Wildfly utiliza una caché de llamada simple. Este es un caché que no *passivation*. Esto significa que todos SFSB se mantienen en la memoria sin ninguna interrupción. A no ser eliminado por la aplicación o su entorno de tiempo de espera, estos pueden provocar una excepción fuera de la memoria en casos extremos. El siguiente comando CLI muestra la configuración por defecto:

```
/subsystem=ejb3:read-attribute(name=default-sfsb-cache)
{
   "outcome" => "success",
   "result" => "simple"
}
```

Es muy recomendable que cambie la configuración para permitir la *passivation* de disco en JBoss/Wildfly. Esto puede hacerse usando el siguiente comando CLI:

/subsystem=ejb3:write-attribute(name=default-sfsb-cache, value=passivating)

El umbral que especifica el número máximo de SFSB en la memoria antes de *passivation* se pondrá en marcha (el valor predeterminado es 100000) y se pueden recuperar y configurado con los siguientes dos comandos de la CLI:

/subsystem=ejb3/passivation-store=infinispan:read-attribute(name=max-size)
{
"outcome" => "success",
"result" => 100000
}
/subsystem=ejb3/passivation-store=infinispan:write-attribute(name=max-size, value=50000)

Un tiempo de espera predeterminado para la eliminación de edad SFSB no utilizado no está disponible en JBoss/Wildfly, por lo que es una buena práctica utilizar la anotación `@StatefulTimeout` en el SFSB para instruir al contenedor para eliminar el bean. De lo contrario, se va a mantener para siempre (o al menos para el siguiente reinicio). El ejemplo siguiente establece a 30 minutos:

```
@StatefulTimeout (valor = 30)
```

el mismo atributo y valor establecido en XML usando el descriptor de despliegue se muestra en el siguiente código:

```
<stateful-timeout>
  <timeout>30</timeout>
  <unit>Minutes</unit>
</stateful-timeout>
```

Desactivación de la *passivation* para un SFSB particular

Como JBoss/Wildfly soporta EJB 3.2, sino que también proporciona la posibilidad de desactivar la *passivation* para una SFSB individual. Esto puede ser hecho en código utilizando anotaciones, como se muestra en el siguiente código:

```
@Stateful(passivationCapable=false)
```

El mismo atributo y el valor establecido en XML utilizando el descriptor de despliegue se muestra en el siguiente código:

```
...
<session>
  <ejb-name>MyBean</ejb-name>
  <ejb-class>org.myapp.MyStatefulBean</ejb-class>
  <session-type>Stateful</session-type>
  <passivation-capable>false</passivation-capable>
</session>

...
```

el uso excesivo de desactivación de la *passivation* puede dar lugar a la aplicación que se ejecuta fuera de la memoria, ya que todos los casos se deben mantener en la memoria.

SFSB individuo puede ser monitoreada mediante la CLI. Ejecute el siguiente comando CLI para enumerar los atributos disponibles:

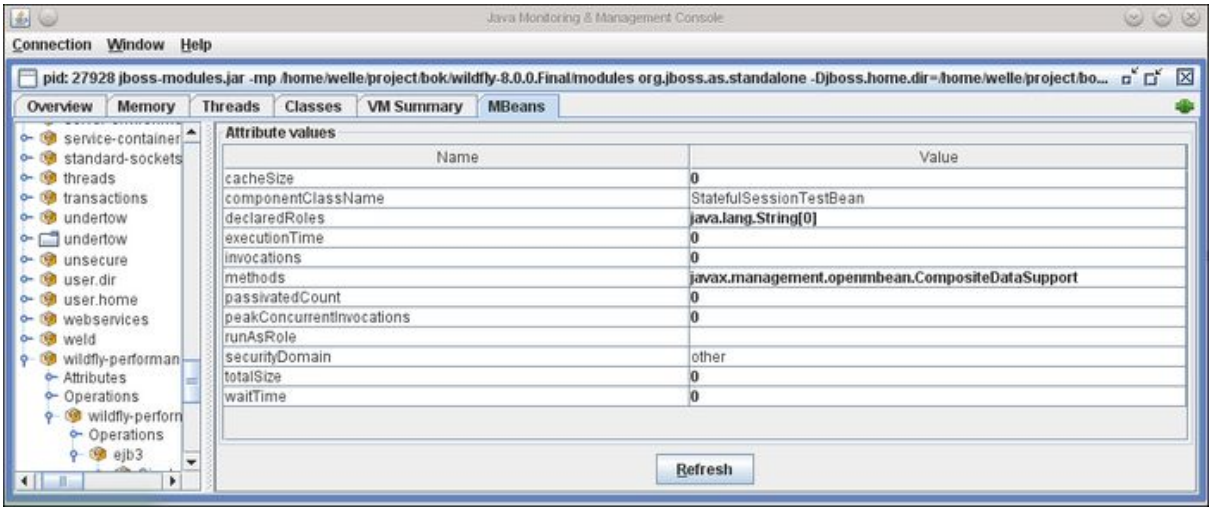
/deployment=MyEAR.ear/subdeployment=MyEJB.jar/subsystem=ejb3/stateful-session-bean=StatefulSessionTestBean:read-resource-description

Los atributos que pueden ser de interés para el seguimiento son los siguientes:

- total-size
- invocations
- cache-size
- peak-concurrent-invocations
- passivated-count

A JMX MBean con el ObjectName

`jboss.as:deployment=MyEAR.ear,subdeployment=MyEJB.jar,subsystem=ejb3,stateful-session-bean=StatefulSessionTestBean`, también está disponible para el seguimiento y se presenta utilizando JConsole, como se muestra en la siguiente captura de pantalla:



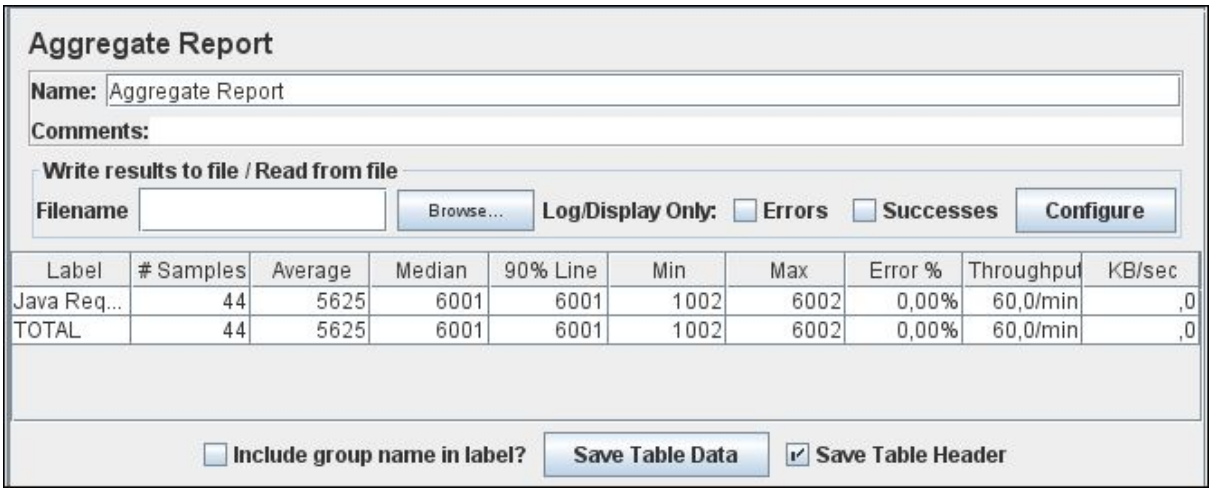
JConsole mostrando la estadística MBean para un SFSB

Optimizando Singleton Session Beans

Parte de la idea básica de beans de sesión es que el contenedor es responsable de no permitir llamadas simultáneas a la misma instancia. Esto ayuda en la construcción de sistemas robustos como el programador no tiene que producir un código seguro para subprocesos.

Una desventaja de esto es que la solidez a menudo puede conducir a problemas de rendimiento. Esto es especialmente común cuando se utiliza Beans de sesión Singleton; ya que sólo hay un caso, puede convertirse fácilmente en un cuello de botella si se deja con su comportamiento predeterminado.

Esto se demuestra en el caso de prueba siguiente. Aquí hay 10 hilos de cliente que ejecutan la realización simultánea de 10 llamadas a un método de Singleton Bean de sesión que se lleva a 1 segundo para procesar antes de regresar. El resultado es un rendimiento de sólo 1 / segundo (60 / min) como se ve en la siguiente captura de pantalla del informe de JMeter:



El resultado de JMeter para un Singleton EJB

Si se estudia los logs de JBoss/Wildfly es probable que vea unos tiempos de espera, así, ya que tomó el cliente más que el valor predeterminado de 5 segundos para obtener una instancia.

Ajuste mecanismos de bloqueo y los tiempos de espera

Mediante el uso de la anotación @Lock, un desarrollador puede instruir al contenedor EJB para aplicar de lectura apropiado o escribir cerraduras más eficaz, como se muestra en el siguiente código:

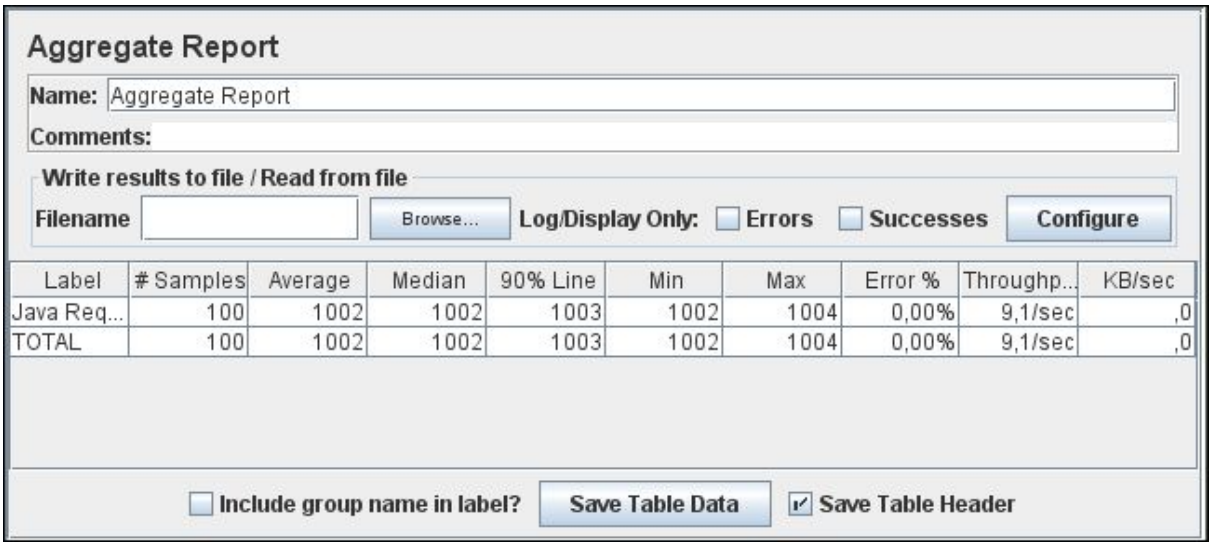
```
@Singleton
@AccessTimeout(value=60, timeUnit=SECONDS)
@Lock(READ)
public class ExampleSingletonBean {
  private String info;

  public String getInfo() {
    return info;
  }

  @Lock(WRITE)
  public void setInfo(String info) {
    this.info = info;
  }
}
```

Este bean ahora utilizar un bloqueo de lectura para todos los métodos, con la excepción del método SetInfo, que tendrá un bloqueo de escritura. El tiempo de espera para conseguir el acceso se establece en 60 segundos.

Que ejecuta el mismo caso de prueba contra un Singleton Bean de sesión con bloqueo de lectura habilitada nos da un rendimiento de casi el 10 por segundo **(9,1**/**seg**), como se muestra en la siguiente captura de pantalla:



El resultado de JMeter de un Singleton EJB con el cierre sintonizado

Container Managed Concurrency vs Bean Managed Concurrency

Cuando el contenedor EJB es responsable del manejo de concurrencia (con la ayuda opcional de la anotación @Lock), se le llama **Container Managed Concurrency (CMC).**También es posible que un programador para tomar el control total cambiando el *bean* a utilizar **Bean Managed Concurrency (BMC)**mediante el uso de las siguientes anotaciones:

```
@ConcurrencyManagement(BEAN)
@Singleton
public class ExampleSingletonBean {
  // Threadsafe code
}
```

Nota

Tenga en cuenta que cualquier código en este bean ahora realmente debe ser *Thread-Safe*. Sin la ayuda viene dado por el contenedor.

Seguimiento

Cada Singleton Bean de sesión se puede controlar mediante la CLI. Ejecute el siguiente comando para listar los atributos disponibles:

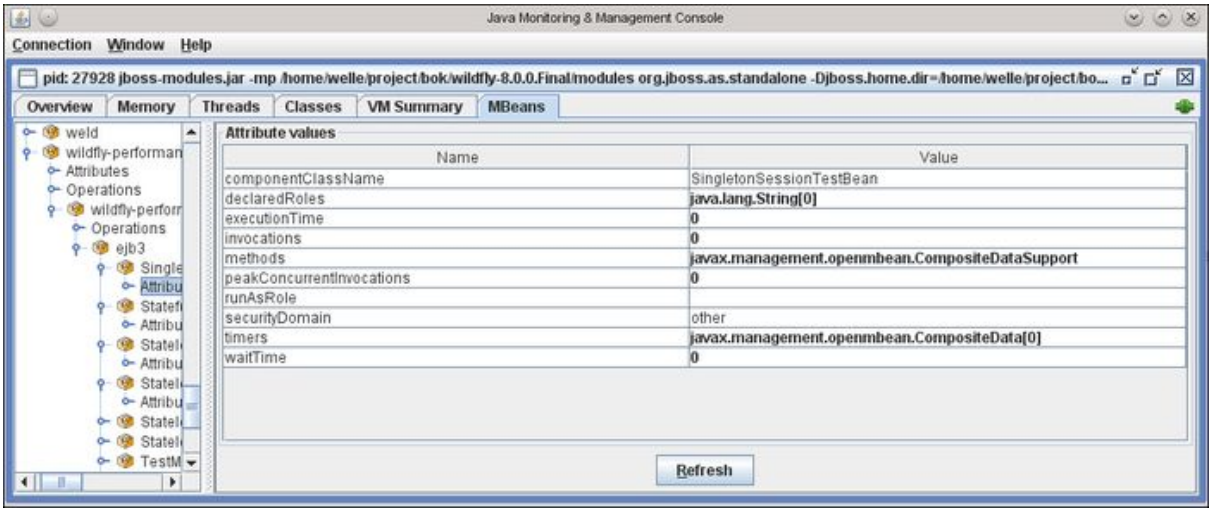
/deployment=MyEAR.ear/subdeployment=MyEJB.jar/subsystem=ejb3/singleton-bean=SingletonSessionTestBean:read-resource-description

Los atributos que pueden ser de interés para el seguimiento son los siguientes :

- peak-concurrent-invocations
- invocations

a JMX MBean con el ObjectName

jboss.as:deployment=MyEAR.ear,subdeployment=MyEJB.jar,subsystem=ejb3,singleton-bean=SingletonSessionTestBean, también está disponible y se pueden ver en la siguiente captura de pantalla JConsole :



El MBean Estadísticas de Singleton Bean se muestra con JConsole

Optimización de Message Driven Beans

MDB es un componente de Java EE que procesa los mensajes de forma asíncrona, a menudo de un de entrada **adaptador de recursos (RA).**El adaptador predeterminado es el proveedor JMS incorporado en JBoss/Wildfly.

Tip

Como JMS se utiliza a menudo junto con los MDB, el ajuste adecuado del proveedor de JMS y los destinos conectados también es muy importante para el rendimiento general. Más acerca de cómo ajustar el proveedor de JMS JBoss/Wildfly, sus mensajes y destinos serán abordados en un capítulo próximo.

Como los mensajes pueden ser procesados simultáneamente, las instancias MDB necesitan ser agrupados en la misma forma que, por ejemplo, un SLSB. El tamaño del pool por

defecto para MDB en JBoss/Wildfly es 20. Esto se puede verificar y modificar usando los siguientes comandos de la CLI:

 # Get the max pool size
/subsystem=ejb3/strict-max-bean-instance-pool=mdb-strict-max-pool:read-attribute(name=max-pool-size)
 # Set the max pool size
/subsystem=ejb3/strict-max-bean-instance-pool=mdb-strict-max-pool:write-attribute(name=max-pool-size, value=30)

Es fácil llegar a la conclusión de que este número corresponde al número de listeners con AR en un destino JMS también. Esto significa que un MDB que escucha los mensajes de una cola JMS haría que el servidor de aplicaciones configurado 20 listeners a esa cola. Esto no es realmente el caso. Estos listeners se configuran por separado con los parámetros AR-específica.

En JBoss/Wildfly, el adaptador por defecto es *HornetQ* (que es el proveedor de JMS), y se utiliza la propiedad *maxSession* (el valor por defecto es 15), que puede ser ajustado en la anotación

```
       @ActivationConfigProperty annotation in MDB, as shown in the following code:
@MessageDriven(activationConfig = {
       @ActivationConfigProperty(propertyName   =   "destinationType",   propertyValue   =
"javax.jms.Queue"),
        @ActivationConfigProperty(propertyName    =    "destination",    propertyValue    =
"queue/testQueue"),@ActivationConfigProperty(propertyName  =  "maxSession",  propertyValue  =
"20"),
       @ActivationConfigProperty(propertyName   =   "acknowledgeMode",   propertyValue   =
"Auto-acknowledge") })
public class TestMDB implements MessageListener {
  public void onMessage(Message message) {
...
       }
}
```

Cada MDB puede ser controlados mediante la CLI. Ejecute el siguiente comando para listar los atributos disponibles:

/deployment=MyEAR.ear/subdeployment=MyEJB.jar/subsystem=ejb3/message-driven-bean=TestMDB:read-resource-descriptiondescriptiondescriptiondescription

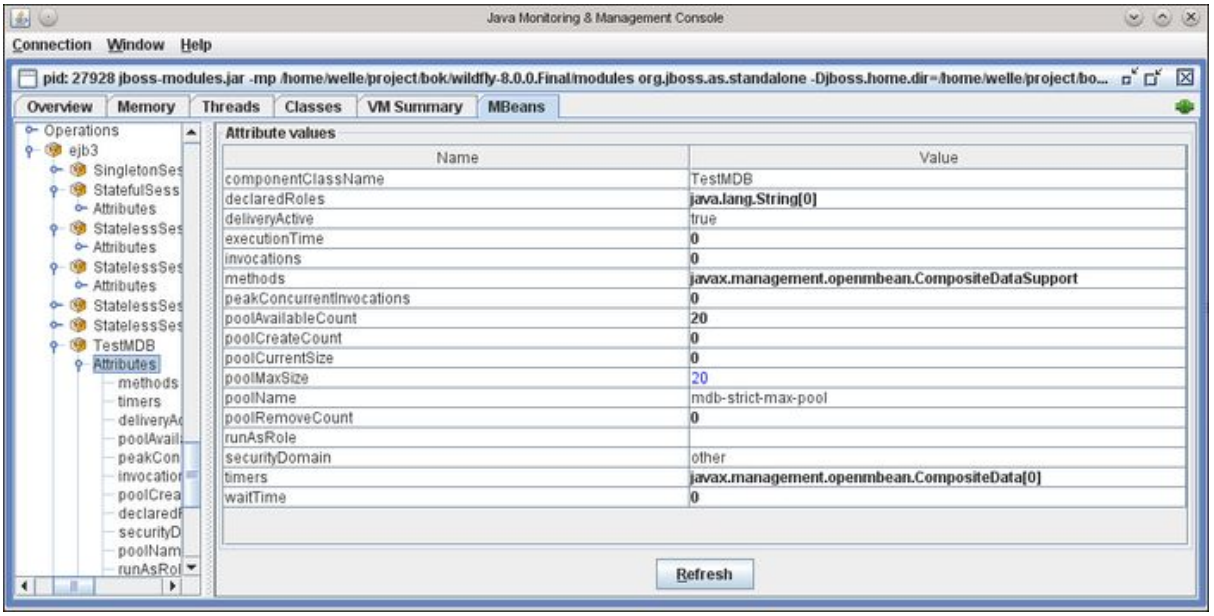
Los atributos que pueden ser de interés para el monitoreo son de la siguiente manera:

- invocations
- peak-concurrent-invocations
- pool-available-count
- pool-current-size
- pool-create-count
- pool-remove-count

al igual que los otros tipos de bean, hay un JMX MBean con el ObjectName

jboss.as:deployment=MyEAR.ear,subdeployment=MyEJB.jar,subsystem=ejb3,message-driven-bean=TestMDB.

La siguiente captura de pantalla visualiza el resultado utilizando JConsole:



Estadísticas MBean para MDB se muestra con JConsole

Un escenario común es que la aplicación necesita para procesar los mensajes no sólo de forma asíncrona, sino también en un estricto orden serial. La solución es a menudo para configurar el parámetro maxSession a 1, haciendo así un "singleton MDB". Esto es, por supuesto, malo para el rendimiento y sería mucho mejor para rediseñar la aplicación para permitir que los mensajes se procesan en paralelo.

Como el MDB a menudo se utilizan para tareas de larga duración, recuerda lo que hablamos acerca de las transacciones. Hacer que el tiempo de ejecución de transacciones

lo más corto posible, tal vez mediante la división de una tarea larga en varios más pequeños.

Tip

Si la pérdida de mensajes es aceptable en caso de fallas, una alternativa al uso MDB podría ser EJB asíncronos para mejorar el rendimiento.
