

解決 Memory leak 有效掌握 Resource

作者：倍力資訊第二事業部 產品技術顧問
陳金生 先生

一般在 3-tier 架構下的應用系統，最常見的問題就是 Out of Memory(記憶體不足)，或 Memory leak(記憶體洩漏)的狀況，最後往往導致 Application Server 失效與系統 Crash，讓管理人員必需常常守候在 Server 旁邊，注意它關心它系統狀況與運作情形避免它 Crash。

而本專題報導則是針對此類型問題的發生，從 JVM 其基本架構開始說明，並採用問答與實例的方式進行說明解釋，並且提供檢查的項目說明，可藉由這些項目自我檢測，以避免發生 Memory leak 有效掌握 Resource。

何謂 Out of Memory ? Memory leak ?

Out of Memory 定義

記憶體不論 Java heap 或 Native Memory 是不足以提供給元件使用。

Memory leak 定義

元件的記憶體使用不論在 Java heap 或 Native Memory 中持續成長，最後導致發生 Out of memory 的情況。

何謂 Java heap, Native memory and Process Size

要了解 Memory leak 問題的發生，首先先了解幾個 JVM 的記憶體管理重要的名稱：

Java heap

這是 JVM 用來配置 Java objects 的記憶體，Java heap 記憶體大小是透過命令執行列中下的參數-Xmx 設定的。假如最大的 heap size 沒有定義，那麼它的大小限制將由 JVM 視當時情況如機器的實體記憶體與剩下可用的記憶體決定。因此一般都建議要設定最大的 Java heap 的值。

Native Memory

這是 JVM 用來它內部運作的記憶體，Native Memory Heap 將是會被 JVM

使用，而它的大小取決於產生的程式碼、產生的 thread、GC 時用於保存 java object 資訊與產生或最佳化程式碼時的暫存空間

假如它是 Third party 的 native module，它將可能使用 Native Memory。例如：native JDBC driver 就是配置 Native Memory。

Native Memory 的最大值是會受限於任何一 OS 的 virtual process size 與已經被參數-Xmx 指定給 Java Heap 的記憶體大小。例如：假如應用系統能夠總共配置 3GB，並且假如最大的 Java Heap 大小為 1G，那麼 Native Memory 最大值可能接近 2GB。

Process Size

Process Size 將會是 Java Heap、Native Memory 與被用於已載入執行與函式庫的記憶體的加總，在 32 位元的作業系統，一個處理程序虛擬定址空間能夠到 4GB；若超過 4GB，作業系統的核心將會預留一部份給它自己使用(一般是 1~2GB)。那麼剩下的就是給應用系統。

Windows：假設有 4GB 的記憶體，預設給應用系統使用最大 2GB 與而另外的 2GB 是給核心使用。儘管如此，在一些不同版本的 Windows，可以用 3GB 參數切換這個比例值，讓應用系統可以得到 3GB 的記憶體。詳細可參考 Microsoft 網站，地址：

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/ddtools/hh/ddtools/bootini_1fcj.asp

Red Hat AS 2.1：應用系統可以使用到最大記憶體大小為 3GB

其他的作業系統，請參考該作業系統文件做設定。

Process 定址空間與實體記憶體的不同

每一個 Process 都擁有自己的定址空間，在 32 位元作業系統，這個定址空間是介於 0 到 4GB 之間。這是機器上獨立的 RAM 或 Swap Space，機器上全部的實體記憶體是同一台機器上的 RAM 與 Swap Space 的加總，所有執行中的 Process 分享這個實體記憶體。

Process 的記憶體定址是虛擬的。作業系統核心對應這虛擬位址到實體位址。實體位址指到實體記憶體中的某一個位置。在一台機器上任何特定時間所有被正在執行的 Process 的虛擬記憶體，其加總不能超出在同一台機器上全部實體記憶體大小。

為何會 Out of Memory 問題發生？在這個狀況發生

JVM 會做什麼處置？

Out of Memory in java heap

假如 JVM 無法在 java heap 取得記憶體來配置更多的 java objects，JVM 會丟出 java out of memory 錯誤，JVM 無法配置更多的 java objects 假如 heap 是塞滿了正在使用的 objects 與 java heap 無法再擴展。

在這個狀況下，在丟出 java.lang.OutOfMemoryError 錯誤訊息後，JVM 會讓應用系統決定要做什麼。例如：應用系統自行監控這個錯誤並且決定在那一個的模式下停止執行，或者不管這個錯誤。假如應用系統不處理這個錯誤，那麼 thread 會丟出這個錯誤訊息並且停止執行離開 JVM(假如使用 java thread dump，你將不會看到這個 thread)。

WebLogic Server 在這個狀況，假如它是由一個 execute thread 丟出，這個錯誤將會被監控並且會做記錄。假如這是連續不斷被丟出來，那麼 core health monitor thread 會停止 WebLogic Server 運作。

Out of Memory in native heap

假如無法 native memory 配置到記憶體空間，則 JVM 丟出 native out of memory，這通常發生在 Process 達到作業系統 Process size 的限制或是機器執行超出 RAM 與 Swap Space 加總。

當這個發生時，JVM 會處理 native out memory 情況，記錄訊息說明它執行到 out of native memory 或無法取得記憶體並且離開。假如 JVM 或任何被載入的 module(像是 libc 或是一個 Third party 的 module)無法處理這個 native out of memory 狀況，然後作業系統將會傳送一個 sigabrt 訊息給 JVM，這樣將會使 JVM 停止離開。通常 JVM 將會產生程式碼檔當它取得 sigabrt 訊號。

處理此問題步驟

Java Out of Memory

1. 收集與分析 verbose gc 的錯誤訊息輸出

- 將 “verbosegc “參數加入命令提示列啟動 Server，這將會將 GC 的活動資訊顯示在標準輸出/輸入，轉到 stdout/stderr 的檔案中。執行應用程式直到問題再次產生。
- 確定在 java out of memory 之前，JVM 做如下內容：

- **Full GC run:**

執行 full GC 與所有 soft/weak/phantomly reachable 的物件能被移除與這些空間能被回收，在下面網址可以找到更多不同等級物件消耗細項說明：

<http://java.sun.com/developer/technicalArticles/ALT/RefObj>

你能檢查 full GC 在 out of memory 訊息之前做，接下來的一個訊息顯示當 GC 已經完成(訊息的格式是依照 JVM 定義，檢查 JVM 的 Help message 以了解訊息格式)：

```
[memory ] 7.160: GC 131072K->130052K (131072K) in 1057.359 ms
```

下面是上面格式的說明(注意：相似格式將會被使用在遍及這份文件)：

```
[memory ] <start>: GC <before>K-><after>K (<heap>K), <total> ms
```

```
[memory ] <start> - start time of collection (seconds since jvm start)
```

```
[memory ] <before> - memory used by objects before collection (KB)
```

```
[memory ] <after> - memory used by objects after collection (KB)
```

```
[memory ] <heap> - size of heap after collection (KB)
```

```
[memory ] <total> - total time of collection (milliseconds)
```

然而，這無法使用訊息推斷 soft/weak/phantomly reachable objects 被移除。假如 GC 演算法是 generational 演算法，你會看到 verbose 輸出像這樣：

```
[memory ] 2.414: Nursery GC 31000K->20760K (75776K), 0.469 ms
```

上述是 nursery GC(或 young GC)循環將會提升正在執行的 objects 從 nursery(或 young space)到 old space。這個循環不是重要的分析，更詳細項目有關 generational 演算法可以在 JVM 的文件中找到；假如 GC 的循環不是發生在 java out of memory，這樣就可能是 JVM 的 bug。

● Full compaction:

保證 JVM 做適當的緊密程度工作，記憶體沒有碎裂，能防止 large objects 被配置而且引發一個 java out of memory 錯誤。Java Objects 需要連續的記憶體區塊，假如沒有空的記憶體區塊，那麼 JVM 將無法配置一個 large objects，它將無法符合任何一個空的區塊。在這個情況下 JVM 將會做 full compaction，這樣才會有更多連續記憶體區塊能夠符合容納 large objects。

Compaction 工作包含移動 objects(data)從 java heap memory 一個區堆到另一個與更新 references 到這些 objects 的指定的新的位置上。JVM 將不會重排所有 objects，除非這是需要的。這是減少 GC 循環的暫停時間。

我們能否透過 verbose gc 訊息檢查出記憶體碎裂的 java out of memory。假如我們看到輸出像下列所示，即使仍有空的 java heap 而 out of memory 還是會發生，是因為記憶體碎裂的原因。

```
[memory ] 8.162: GC 73043K->72989K (131072K) in 12.938 ms
[memory ] 8.172: GC 72989K->72905K (131072K) in 12.000 ms
[memory ] 8.182: GC 72905K->72580K (131072K) in 13.509 ms
java.lang.OutOfMemoryError
```

以上的情況你能夠看出 max heap 是被設定 128MB 與 JVM 丟出 out of memory 當實體記憶體使用只有 72580K，Heap 使用率只有 55%。因此，即使當有 45%free heap 的時候，在這個案例記憶體碎裂的影響仍會丟出 out of memory。這是 JVM 的 bug 或是限制。你必需聯絡原廠請求協助。

2. 假如 JVM 運作是正常的

假如 JVM 運作是正常的(的在上述所提及的動作)，那麼 java out of memory 可能是應用程式的問題。這個應用程式可能不斷洩漏一些 java memory，可能是會引發這個問題。或者是應用程式使用過多的 objects 它需要更多的 heap memory，以下是可以在這個應用程式中做檢查的：

- 應用程式的 caching：

假如應用程式在記憶體中 caches java objects，那麼我們將需要確定這個 cache 是否一直在成長；可能需要在 cache 中限制 objects 的數量。我們能試著降低這個限制看看是否它會降低 java heap 使用量。JAVA soft references 像 softly reachable objects 一樣使用 data caching，當 JVM 執行發生 out of heap 時，是允許被移除。

- 執行過久的 objects：

假如在應用系統裡有在 Heap 中存在過久的 objects，那麼我們可以儘可能地試著降低存在的 objects。例如：調校 HTTP session timeout 將能幫助更快回收無效 session objects。

- Memory leaks：

一個 memory leak 的範例是在應用系統中使用 database connection pools。當使用 connection pools，JDBC statement 與 resultset objects 必需確定最後有被關閉。這基於事實，這由於從 pool 中的 connection objects 使用呼叫 close()，將簡單 connection 傳回到 pool 以提供重新使用，若沒有真正關閉 connection 與關聯到的 statement/resultset objects。

- 增加 java heap：

我們也能試著增加 java heap 假如可能的話看是否能解決問題。

3. 假如不是屬於前兩者的狀況：

那們我們需要使用 JVMPi(JVM Profiler Interface)基礎 profiler 像 Jprobe 或 Optimizelt 去找出那些 Objects 是佔住 java heap，profilers 也從這些對象正被建立的地方，在 java code 裡的細節地方上。這份資料不包括在每 profiler 上的細節。profiler 文件可能提到有關如何設定與啟動應用系統與profilers。通常，基於JVMPi的profilers 會有過多負擔與大大降低應用系統的效能。因此，在上線環境裡使用這些profilers 是不適當的。

For Native OOM Problem:

1. 收集下列資訊

- `-verbosegc` output to monitor the java heap usage.

這將能幫助了解應用系統的 java memory 需求。應用系統它可能會需要獨立使用的實際 java heap，在 JVM 啟動時預留最大的 heap 設定(使用 `-Xmx` 參數在 java 命令提示列)與在其他用途是不允許預留記憶體。

在這個 Jrockit 的案例，使用 `-verbose` 取代 `-verbosegc` 如同在 GC 資訊中增加給 codegen 資訊。

- 記錄 Process virtual memory size 從應用系統被啟動時，開始在 JVM 執行到 out of native memory；這將能幫助了解不管是 process 真的達到作業系統的大小限制。

在 Windows 的案例，遵照下列程度監控 virtual process size:

在開始執行，輸入“perfmon”並且按 OK

在 Performance 圖形上方按下 “+” 按鍵

選擇下面項目：

效能物件: Process (不是預設的 processor)

從清單選取計數器: Virtual Bytes

從清單選擇例項: Select the JVM (java) instance

按 “新增”，並且 “關閉”

在 Unix 或 Linux 中對於 PID 而言，virtual memory size 能夠使用這個命令看到：`ps -p <PID> -o vsz.`

在 Linux 中每一個 java thread 都會有單一個 JVM instance 以 process 方式顯示，假如我們選擇一個 root java process 的 PID，這個 root java process 可以使用 `ps` 命令加上 `-forest` 參數能夠被發現，例如：`ps -IU <user> --forst` 將會顯示某一特定使用者所有 process 的 ASCII 的樹狀結構顯示，你還可以從這樹狀結構找到 root java。

2. 機器上可用的記憶體

假如機器 RAM 與 swap space 是不足，那麼作業系統將無法提供更多的記憶體給 process，那麼也可能造成 out of memory 的結果，確定同機器上的 RAM 與 swap space 在硬碟空間的加總是足夠執行所有的 Process。

3. Java heap 調校

假如 java heap 的使用都在 max heap 之內，那麼可以考慮降低 max heap 的大小，以提供更多的 native memory 給 JVM 使用。這並不是一個解決方案，但是一個測試的變通方法。因為作業系統限定 process 大小，我們需要在 java heap 與 native heap 之間達到一個平衡。

4. JVM 的 Native memory 使用量

在所有的 classes 被載入 JVM 的 Native memory 使用量是將可預期會趨於平穩，並且方法已被呼叫(code 的產生已經結束)。應用系統通常發生在最初的幾個小時，在那之後，JVM 可能在執行時載入 class，code 的最佳化只需要很小的 native memory。

爲了減少問題，試著取消執行時最佳化並且檢查是否有何不同。

- 在這個 jrookit 案例，參數-Xnoot 能夠取消執行時最佳化。
- 在 SUN 的 JVM，參數-Xint 將強制 JVM 執行在 interpreted mode(no code generation)。

假如執行時 native memory 使用量仍持續成長，那麼問題就可能是 native code 這邊發生。

5. 應用程式中 Third party 的 native modules 或 JNI code

檢查不論你是使用 Third party 的 native module 像是 database driver，這些 native modules 可能也會配置 native memory 並且漏洞可能就是這些 modules。基於縮小問題，你能試著重建問題在不使用協助廠商的 driver，例如：你能使用 pure java drivers 取代 native database drivers。

檢查你的應用系統是否使用 JNI 的程式碼，這也可能造成 native memory 洩漏，並且你也可以嘗試執行應用系統儘量不使用 JNI 程式碼。

6. 假如照上述步驟仍無法找出 native memory 問題

那麼你就得向 JVM 廠商取得能偵測 native memory 配置的特別的版本並且更多有關 JVM 上有關記憶體洩漏的訊息。

結論

系統開發時往往問題發生 out of memory，或記憶體一直成長無法透過 GC 回到平均值時，這往往不知道該如何處理此問題所在，到底是應用系統問題，還是 JDK 發生的問題，以上的這些是針對有關 Memory leak 問題進行說明的了解，與問題排解原則加以說明，期望日後再碰到此問題能有所參考依據與檢測的方式，以讓此類問題能夠不再發生。

參考資料

Investigating Out Of Memory / Memory Leak Problems :

http://support.bea.com/support_news/product_troubleshooting/Investigating_Out_of_Memory_Memory_Leak_Pattern.html