伺服器壓力測試軟體之效能評估

鄭為民　　李育軒
東吳大學資訊管理學系
wjeng@csim.scu.edu.tw

摘要

伺服器壓力測試(stress test)是目前常用來評估伺服器效能的方式，利用短時間的大量存取服務，造成伺服器處於高負載的目的，在此情況下可以顯示效能的極限。本篇文章使用一套伺服器測試軟體 Apache JMeter 為例，調校此測試軟體的執行環境，並修改程式的記憶體配置方式，以求能夠執行時能夠更穩定及快速的產生測試結果。藉由 JMeter 模擬多使用者同時連線所產生的測試結果，評估伺服器的效能瓶頸。

關鍵詞：壓力測試、JMeter。

Abstract

Server performance test is an important task of evaluating consistent and rapid server services for users. Therefore, stress test software has been commonly used to generate concurrent connections to the server service to simulate real-world situations. This study uses server testing software - Apache JMeter in order to collect and analyze the test results for better tuning of the critical software. Executing environment parameters and memory allocation schemes are adjusted to overcome the performance bottleneck for reliable test results.

Keywords: Stress test, JMeter.

1. 簡介

由於網路的發展與需求日漸增加，提供的各式服務需求也跟著提高，伺服器的負載也越大，使用高效能伺服器不足以完全解決問題，而必須注重整體的規劃。使用壓力測試來評估伺服器效能是必備的，藉由軟體模擬大量連接數，提高伺服器的使用率，觀察服務處理量可承受的最高範圍，以及在此環境下，伺服器運作時可能產生的問題。壓力測試的結果成為伺服器效能評估以及效能調校很重要的依據，可以從中找出伺服器效能的瓶頸，再針對軟體部分的支援，或者硬體架構的調整來增進伺服器服務的效能。

Apache JMeter[2]是一個開放原始碼的伺服器測試軟體，具備分散式測試的功能，可使用多台分散於不同位置的電腦來同步測試，再集中統計所有的測試結果。模組化的設計讓使用者可用 plugin 的方式擴充額外的測試或報表元件，且使用圖形介面來配置的測試的流程，輔以可程式化功能，定義複雜的取樣方法。擴充的 plugin 及測試流程和主程式，可運行於各種不同的平台。

本篇文章使用 JMeter 工具為例，來對一個網頁伺服器做壓力測試，分析伺服器的效能，找出伺服器所能承受的範圍。因 JMeter 是 100%由 Java 開發的程式，執行的環境受 JVM 所限制，在進行壓力測試時會有不穩定，甚至當機的情況。

進行壓力測試之前會先測試 JVM 的最佳化對程式執行效能的影響，主要是針對記憶體管理機制方面的調整，再提出以修改 JMeter 執行緒所需內容物件的使用方式，使用 memory pool 的概念，讓物件盡可能的重複利用，不去配置一個新的物件，減少 heap 空間的持續消耗使用。降低垃圾回收(GC)產生的次數，相對也減少 GC 所帶來的程式中斷[5]，提升執行時的效能。最後根據 JMeter 壓力測試所得的數據，分析伺服器的效能，找出伺服器所能承受的範圍。

其他章節分敘如下，第 2 節介紹伺服器測試方法與問題，以及 JVM 的分代收集器，第 3 節描述實驗環境檢查與實驗方法，第 4 節根據實驗所得的數據進行分析，並探討 JMeter 最佳化及修改 JMeter 機制的影響，第 5 節為結論與未來進一步研究的方向。

2. 相關研究

系統測試有很多不同的流程，在系統推出之前，會經過不斷地測試，直到功能穩定且正常執行。待功能等方面的正常測試有了整體評估後，則會針對較特殊的狀況來做測試，嘗試大量且頻繁的測試此系統的功能，使用一些較不正常的使用操作測試，目的是造成系統運行時的不穩定，甚至停止服務，這種測試稱為壓力測試。壓力測試用來測試系統在資源不足時，以及大量使用的情況下，系統的異常反應，找出所能承受的最大容許程度，從測試的結果可得知，不超過此臨界點的話，可保證系統的正常運作。

2.1 伺服器效能測試

軟體老化(software aging)的問題，通常出現於系統長時間的使用，造成性能的下降以及系統資源不足 [6]等問題。要找出老化的問題，可使用不同

的參數，長時間測試之後，從測試結果中發現問題的所在。測試參數的選擇也是困難的一件事 [4]，例如具有多個下拉式選單及選項的網頁表單，其中一種組合大量存取時會引發錯誤，但測試組合可能有上千種，因此這個是進行壓力測試時比較麻煩的地方。

根據壓力測試所得到的結果，可以做進一步的分析。不同的伺服器軟體，對於資源請求的種類有不同的處理效能 [12]，可根據測試結果，分別使用效能較好的伺服器，服務這些資源請求，提高服務效能。現在網頁跟資料庫伺服器大多是分開的，當連線時請求一個頁面資料，網頁伺服器會去跟資料庫伺服器取得資料再回傳結果頁面。若直接進行壓力測試，觀察連線的成功率以及處理量的話，可能會不太準確，沒辦法正確評估效能瓶頸在哪一台伺服器上。因此可以在兩台伺服器內部使用系統效能監控工具，搭配壓力測試一同使用 [1]，這樣較能看出效能瓶頸是在哪一台伺服器。

一般伺服器的壓力測試，使用漸進式增加最大連線數 [14]，或是增加每秒連線的數目[3] 的方式去測試，壓力測試結果會著重在回應時間、錯誤率、處理量及頻寬大小方面的數據。另外也可以去測試使用持續連線(keep-alive)與不持續連線的效能差異 [13]。根據連線數及連線頻率的增加，可以找出伺服器的瓶頸所在，進而改善效能及增加服務使用人數。

執行 JMeter 來做壓力測試時，首先會遇到系統資源不足的問題，導致 JMeter 因為記憶體的垃圾回收機制正在進行而停頓太久，甚至還沒執行結束之前就當機(crash)，因此需要對 JMeter 的 JVM 環境去做調校，才能進行穩定的壓力測試。

2.2 HotSpot VM 記憶體管理

在記憶體管理方面，JVM 會配置一塊 heap 作為記憶體配置的使用，通常 JVM 會在記憶體不足時，會使用 GC 試圖回收 **錯誤! 找不到參照來源。** 不再被任何參考使用的記憶體。比較重要的是，GC 發生時 Java 程式會中斷執行(Stop-the-world pause)，等待 GC 結束才會再繼續執行，嚴重影響程式執行的效能。因此配置記憶體的速度越快，以及 GC 所花的時間越短，引起的中斷次數越少[7]。程式執行時的資源使用及運算可以連續執行，效能會比較好。

HotSpot 的垃圾回收器使用多種 GC 演算法組合，以分代收集(Generational Collection)為基本演算法，把 heap 分為兩個部分，分別是年輕代(Young/New Generational)及年老代(Tenured/Old Generational)，其中年輕代是由 eden 加上兩個 survivor space 組成。物件初始化時會在 eden 分配記憶體，經過 GC 後依然被參考的物件會移到 survivor space，其中 survivor space 用來當做與 eden 區域的交換空間來交替使用；當物件存放於 survivor space 經過一段時間還沒被回收，就會把物件移動到年老代。除此之外還有永久代(Permanent Generation)，存放 class 和 method 資訊，這些資料載入後就不太會再變動，因此這個地方的大小不會經常改變。

分代收集器(Generational Collector)為了避免額外的工作，在觀測物件存活的生命週期中發現，大部分的物件，剛被初始化不久後就不再使用，代表此時物件就可被回收，例如迭代物件，只在這單一迴圈週期有被使用。

而有些物件從初始化到結束才會回收[11]。為了物件存活的時間長短不同的效能最佳化，在年輕代及年老代兩區域有不同的 GC 演算法。年輕代所使用的是次要收集(minor collection)，針對大部分物件短暫的生命週期最佳化，速度非常快；而年老代所使用的是主要收集(major collection)，因要去掃描整個年老代裡面的物件，能不能被參考到，並清除記憶體破碎，因此做一次主要收集所花的時間很久，相對於次要收集約幾十倍以上，所以要儘量減少主要收集發生的機率。

1.4.2 版本的 HotSpot VM 預設用了分代收集器來做 GC，除此之外還有另外三種。每個分代收集器都有加強應用程式的垃圾回收處理量或是較低的垃圾回收時間[9]。表 1 是 HotSpot VM 所支援的四種分代收集器：

A. 預設 GC
B. Throughput GC
C. Concurrent Low Pause GC
D. Incremental Low Pause GC

表 1 分代收集器

	CPU	特性
A	1	使用單一 thread 進行分代的回收。
B	2+	使用多個 thread 進行年輕代的回收，但年老代還是使用單一 thread。
C	2+	減少 GC 造成程式中斷的時間，在回收年老代的同時，還是可以繼續執行程式。
D	1	在回收年輕代的同時，也回收部分年老代的物件。

雖然 Incremental Low Pause GC 把年老代分成幾個部分，在年輕代回收，減少 GC 帶來的中斷時間，但其演算法卻會造成程式執行效率的大幅下降。

2.3 HotSpot VM 選項

一般的 Java 程式不需要特別的參數設定，就可以順利執行，但較為複雜或是大型的應用程式，就需要去設定參數調整，達到系統效能的最佳化。JVM 執行時可選用 Classic VM 或是 Java HotSpot

VM 的 Client VM 及 Server VM。其中 Classic VM 不使用 HotSpot 的技術，因此效能較差[14]；Client VM 及 Server VM 都共用 HotSpot 執行環境的基本功能，除此之外，Server VM 會使用較複雜的最佳化技術來執行 server 端的應用程式，Client VM 則是對 client 端的應用程式做最佳化處理，這些不同的執行環境，讓應用程式在執行 server 端或 client 端程式時有更好的效能表現。

透過 command-line 選項和環境變數設定，可以影響 HotSpot VM 的效能特徵[10]。HotSpot VM 的選項分成標準選項、非標準選項、不穩定且不建議隨意使用的選項，除了標準選項以外，非標準及不穩定的選項是屬於實驗性階段的選項，可以設定的項目也比較多，可能會在未通知的情況下改變，因此每個版本的 JVM 都需要先檢查選項功能是否有所不同。

3. 測試方法

以版本為 1.4.2 的 J2SE 平台為例，測試在多 CPU 環境底下，多 CPU 同時執行 GC，以及調整 JVM 參數對效能的影響。我們希望能夠在測試伺服器效能時，程式運行能夠更加穩定及流暢，可以模擬真實的多使用者同時操作的情況。調校完 JMeter 的效能之後，再對伺服器作壓力測試，看回應時間及錯誤率的數值，找出伺服器的壓力所在。

3.1 實驗測試環境

所使用的硬體為 SGI 伺服器(Silicon Graphics Prism Visualization System)，四顆 1.5GHZ Itanium2 的 CPU，以及 4G 的 RAM 所組成。作業系統使用 Red Hat Enterprise Linux AS release 3 64-bit，Java 版本為 Java HotSpot 64-Bit Server VM 1.4.2_09-b05。

首先從系統方面著手，先測試兩個系統設定：

1. Max Thread 數量：

   分別使用 C 及 Java 去測試系統最大 Thread 數，跟/proc/sys/kernel/threads-max 裡面所設定的有關，跑出來的結果兩者差不多，加上系統正在執行的 Process 數，約是系統設定的數字 15290 左右。再以 ulimit -u 去查看使用者 Process 的使用限制，這邊會影響到執行緒的總數量。

2. Max Open File 數量：

   以 ulimit -n 去查看 Process 的開檔限制，預設是 1024，因為在 linux 底下 socket 是當做 file 處理。為了能夠一次開更多 socket，所以這邊修改成 2000。

3.2 JMeter 程式修改

經過分析 JMeter 的原始碼後，可以看出網頁請求會使用執行緒來接收一次或多次回應。執行緒本身會配置一個物件來存放資料，取樣前後也都會配置物件來存放讀取及寫入的資料，存取回應時會去配置物件的記憶體空間，結束時再去做釋放。當測試進行所需的次數很大時，會很頻繁的去配置及初始化一個物件。因此想法是讓物件重複使用，避免使用 new 的方式去配置新的物件。修改成以 memory pool 的想法，用 stack 來存放物件的參考，把物件 cache 住。當物件用完後，丟回 pool 中，下次需要時，再去 pool 取出，期望這樣做能夠減少 GC 所帶來的負擔，增進程式效能。如下圖 1 所示。

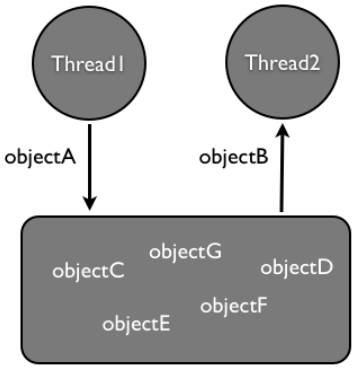

圖 1 Memory Pool

3.3 實驗測試參數

測試對象為 Yahoo 新聞網站，共有 13 種新聞分類，每分類八則新聞，模擬真實情況下，隨機存取分類的一則新聞。使用執行緒來模擬使用者的操作，測試時每秒新增 10 個執行緒，並且每個執行緒要求取樣時都延遲 300ms，而不是連續的請求。

先前對四種分代回收器做測試，測試出來的結果顯示，預設的分代回收器各方面均有不錯的效能，比另外三者來的穩定許多，速度也幾乎都是最快的，因此測試時採用預設的分代回收器。

主要以幾種不同的參數組合來測試修改前後所帶來的改變，測試參數分別有 Heap 的大小，看資源的多寡是否會有影響。執行緒的數量則是測試時所產生的最大執行緒數，每次測試時遞增 100，直到 1200 為止。每個執行緒重複請求次數分別設定 10 與 50，看是否請求的次數越多，會影響效能。另一個測試不限制執行緒請求的次數，因此是一直重複請求，但限制測試的時間為 10 分鐘。參數如下表 2。

表 2 測試參數

	Heap	執行緒	重複次數	時間
A	1GB	100~1200	10	X
B	1GB	100~1200	50	X
C	3GB	100~1200	10	X

D	3GB	100~1200	50	X
E	3GB	100~1800	Forever	10min

4. 實驗結果與分析

測試參數組合的 A、B、C、D 分別是限定請求的次數，比較的是測試完成的時間，看修改前後何者測試所花費的時間較少；而 E 則是限定請求的時間，比較的是連線取樣數、及網站的每秒處理量。同時間的連線取樣數越多，表示測試的效能越好，每秒處理量越高，表示受 GC 的影響較小。最後 GC 所花費的時間越少，表示執行時被中斷的時間越少，效能也越好。

下圖 2 到圖 5 是修改前後，測試開始到結束所花費的時間，X 軸表示執行緒的數目，Y 軸表示秒數。藍色的線表示為原始程式測試所花費的時間，紅色的線表示修改記憶體配置後所花費的時間。

由圖 2 可看出當執行緒數目為 100~1100 時，測試所花費的時間，基本上是大概相同，直到 1200 時，才有大幅度的差異。原因在於年輕代空間滿了要做 GC 來回收物件，沒被 GC 回收掉的物件要移到年老代，年老代的空間不夠而做 GC 耗費太多時間，整個測試時間被拉長。而修改後的程式不會一直去配置新的記憶體空間，因此年輕代的空間不會這麼快滿，進而減少年老代做 GC 的次數，所以測試時間較不受影響。

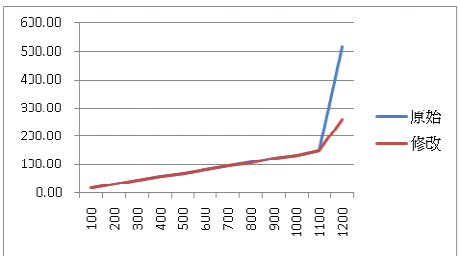

圖 2 執行時間-A

由圖 3 可看出較多的執行緒請求重複次數，對於原本的測試時間影響較為顯著，在執行緒數目 700 時，測試時間開始大幅增加，而修改後的程式則是到 900 測試時間才開始大幅增加，時間也都比原本來的短。至於 1100 以上的話，測試時間超過了 30 分鐘，因此只做到 1000 為止。

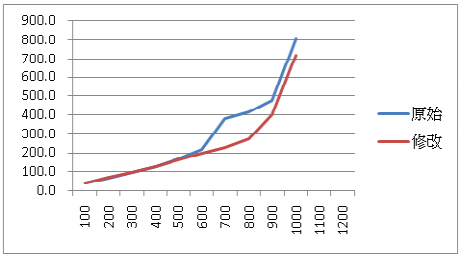

圖 3 執行時間-B

圖 3 及圖 4 的話，因為 Heap 空間較大，所以測試時間的差別並不會太大，圖 4 有較多的執行緒請求重複次數，所以 900 開始測試時間較原本來的短。因此在 Heap 空間足夠時，修改過的程式並不會比原本的程式來的快。

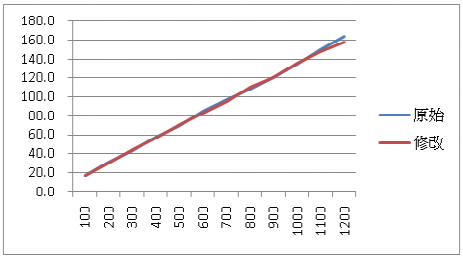

圖 4 執行時間-C

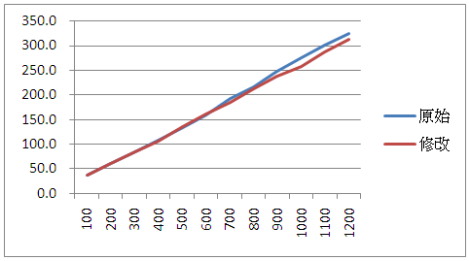

圖 5 執行時間-D

下圖 6 到圖 9 是修改前後，測試開始到結束年輕代及年老代 GC 所花費的總時間，X 軸表示執行緒的數目，Y 軸表示秒數。藍色的線表示為原始程式 GC 所花費的時間，紅色的線表示修改記憶體配置後 GC 所花費的時間。

由圖 6 可看出在執行緒數目 100~1100 時，修改前後差不了多少，而在執行緒數為 1200 時才有大幅改善。雖然在這之前也有減少年輕代 GC 的次數及時間，但就比例而言，影響並不大。而年老代 GC 一次的時間很長，所以才會有很明顯的改善。

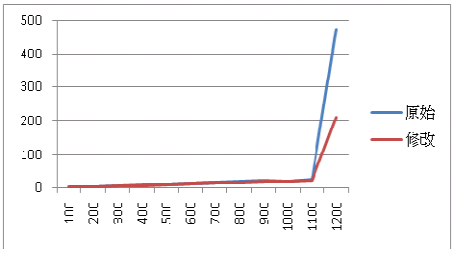

圖 6 GC 時間-A

圖 6 與圖 7 顯示的曲線，基本上與執行時間很類似。圖 6 在執行緒數目 1200 時，程式修改前後，GC 時間減少約 250 幾秒。圖 7 的 GC 時間從執行緒數目 700 開始，程式修改前後，GC 時間平均減少約 60 幾秒，對於執行時間有很大的幫助。

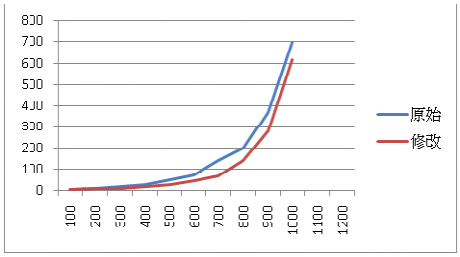

圖 7 GC 時間-B

圖 8 及圖 9 的 GC 時間改善並不是那麼的多，對於程式測試的時間影響有限。圖 8 程式修改前後改善 GC 時間約 4 秒左右，因此圖 4 的測試時間幾乎相同，兩條線幾乎是重疊的。而圖 9 程式修改前後改善 GC 時間約 20 秒左右，所以圖 5 的測試時間有稍微的改進，但不是非常的顯著。

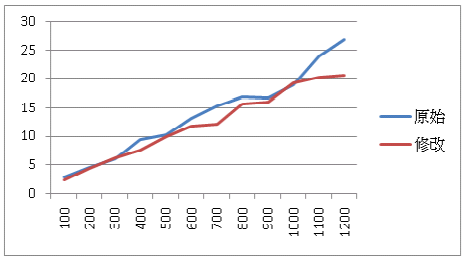

圖 8 GC 時間-C

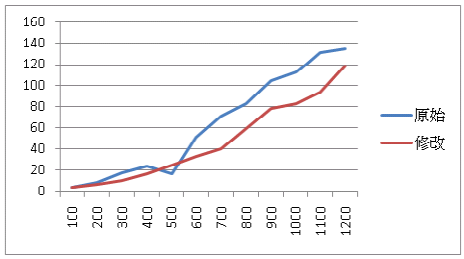

圖 9 GC 時間-D

下圖 10 到圖 12 是以固定一段測試時間來做測試，測試開始到時間結束所得的數據。X 軸都是表示執行緒的數目，圖 10 的 Y 軸的單位是秒數，圖 11 與圖 12 的 Y 軸的單位是個數。藍色的線表示為原始程式測試所花費的時間，紅色的線表示修改記憶體配置後所花費的時間。

由圖 10 中可看出 GC 的時間，平均少了約 40 幾秒。由於 Heap 空間夠大，測試時不會常常因為空間不足，而頻繁的進行年老代的 GC 回收。因此 GC 的時間是跟執行緒的數目成正比。

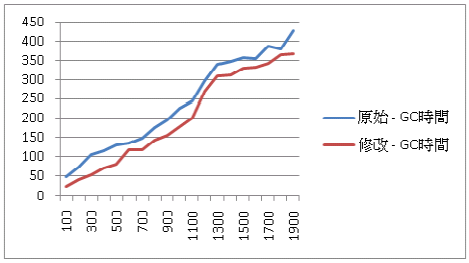

圖 10 GC 時間-E

從圖 11 與圖 12 可看出，執行緒數量約為 1200 以上，修改過後的程式在取樣數，以及網站每秒處理量，都明顯比原始的程式增加許多，可看出因為 GC 的時間減少，所帶來的測試效能增進。

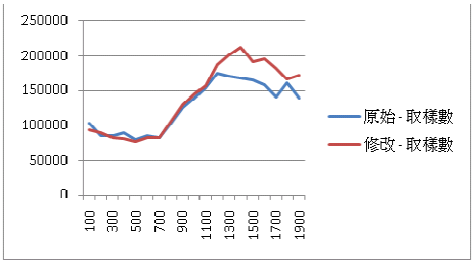

圖 11 取樣數-E

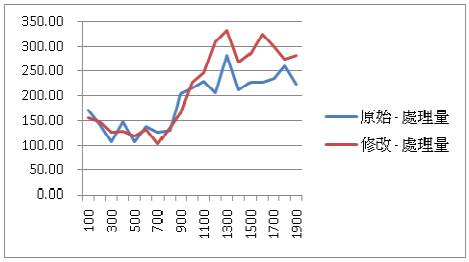

圖 12 處理量-E

JMeter 壓力測試的結果如表 3 所示，也是測試參數 E 的部分數據。取樣數代表這 10 分鐘內總共測試了多少頁面，平均值代表所有連線回應的平均時間(ms)，標準差則是大部分回應時間與平均時間的平均差，錯誤率表示伺服器回應錯誤，處理量則是伺服器每秒所處理的頁面數。這裡只列出修改後程式測試的數據。

表 3 壓力測試數據

執行緒	取樣數	平均值	標準差	錯誤率	處理量
100	93203	304	150.60	29.22%	155.26
300	82056	1790	2805.36	29.35%	127.42
500	77118	2663	4715.93	29.10%	118.78
700	81917	2827	5712.55	32.50%	103.93
900	129371	3339	5902.75	53.22%	166.08
1100	156306	3330	6261.10	56.29%	245.90
1300	201551	4531	8765.81	66.79%	332.07

其中錯誤率一開始就是 29%左右的原因，是因為 Yahoo 網站為了防止同一個 IP 重複做請求所回傳的錯誤，一般正常回應的 HTTP 狀態碼是 200，此錯誤所回應的狀態碼是 999。本實驗經過長時間測試的結果顯示，即使只有開 1 個執行緒，僅取樣 13 次，每個執行序延遲 300ms，還是會收到 999 的回應，這個要經過一陣子，Yahoo 那邊才會停止回應 999 的狀態碼。因此在此情況下測試，會有 29%左右的錯誤率。

根據取樣的平均時間表示，同時間越多的請求，會使得平均回應時間增加。由標準差的值可表示，回應的時間變得較分散，速度較不穩定。而錯

誤率在執行緒數目為 700 以上時明顯增加，伺服器一時之間無法處理這麼多請求，因此更容易造成請求回應失敗。執行緒數目為 100~600 的取樣數都較為相近，而執行緒數目為 700 以上時的處理量卻是持續提高，這是因為 999 的錯誤回應增加了，造成處理能力增加的假象。

由實驗結果可得知，在執行緒為 700 以上時，延遲時間為 300ms 時，伺服器會開始較頻繁的發送錯誤回應。在這 10 分鐘內，扣除錯誤率，所能正常服務的頁面為 54675~73327，每秒約能處理 91.1~122.2 的正常請求。

計算方式如下：

正常服務頁面 = 取樣數*(100%-錯誤率)

每秒服務頁面 =正常服務頁面 / (10*60)

5. 結論

提高壓力測試的正確性及減少測試時間，對於做壓力測試而言幫助很大。為了讓進行壓力測試時，能夠更穩定以及節省記憶體資源，本研究提出修改記憶體緩衝區的配置模式。原本方式是使用完就捨棄，需要時再重新配置一塊緩衝區來使用的行為模式，改為使用快取的方式提供程式使用，使用完就記錄起來，等待下一次的取用。

此方法能夠在負擔很大的測試系統得到顯著的效能提升。因為做壓力測試時需要大量的記憶體，JVM 會因記憶體不足而去做垃圾回收來釋放記憶體空間，而造成長時間的程式中斷。原本的方式會導致大量年輕代物件做 GC 時無法立即被移除，而這些沒被回收的物件就會轉移到年老代，使得主要收集發生的機會變高且執行起來時間更長。使用本研究所提出的修改方法，可以有效重複利用已配置的記憶體資源，減少頻繁的配置及釋放記憶體的行為。進而降低主要收集發生的機會，這樣一來程式中斷的時間少了，執行測試的效能也會變好。

對於本研究所做的壓力測試，主要是以執行緒上限數量來測試瞬間連線數對於伺服器的影響。對於伺服器回應的時間及內容來做分析，可看出伺服器在瞬間連接數越高時，測試數據所出現的增長，與伺服器的效能成反比。從錯誤率可看出伺服器無法處理大量的連線請求，因而回應的錯誤率一直上升。

未來，可以針對物件快取部分去發展模組。目前只針對網頁伺服器測試的效能做改進，因此其他測試的效能並沒有多大改變。若可以使用統一的快取模組，再去對不同的部分做快取的最佳化，可以更有效地提升整體的測試效能。至於 JMeter 本身使用龐大的 heap 空間，在資源不足時容易造成緩慢以及當機。這邊可以去改善記憶體的使用方式，讓程式更加輕巧以及穩定，如此一來可增加壓力測試的極限，測試的結果也能夠更準確。

參考文獻

[1] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani and W. Zwaenepoels, “Bottleneck characterization of dynamic web site benchmarks,” Third IBM CAS, 2002.

[2] Apache Software Foundation. JMeter. http://jakarta.apache.org/jmeter/

[3] G. Banga, P. Druschel, “Measuring the capacity of a web server,” In USENIX Symposiumon Internet Technologies and Systems, pp. 61-71, Monterey, CA, December 1997.

[4] A. Bertolino, “Software testing research: Achievements, challenges, dreams,” 2007 Future of Software Engineering , 2007.

[5] T. Brecht, E. Arjomandi, C. Li, and H. Pham. “Controlling garbage collection and heap growth to reduce the execution time of Java applications,” Transactions on Programming Languages and Systems, vol. 28, 2006.

[6] L. Li, K. Vaidyanathan, K. Trivedi, An approach for estimation of software aging in a web server. computer.org, 2002.

[7] Q. Li, “Java Virtual Machine - Present And Near Future,” Technology of Object-Oriented Languages, TOOLS 26. Proceedings, pp. 480-480, 1998.

[8] C. Lo, W. Srisa-an, J. Chang, “Who is collecting your Java garbage?” IT Professional Vol. 5, Issue 2, pp. 44-50, 2003.

[9] D. Luke, S. Witawas, C. Morris, “An analysis of the garbage collection performance in Sun's HotSpot Java Virtual Machine,” Performance, Computing, and Communications Conference, 21st IEEE International, pp. 335-339, 2002.

[10] Sun Microsystems. Java HotSpot VM Options. http://java.sun.com/javase/technologies/hotspot/vmoptions.jsp

[11] Sun Microsystems. Tuning Garbage Collection with the 1.4.2 Java[tm] Virtual Machine. http://java.sun.com/docs/hotspot/gc1.4.2/

[12] L. Titchkosky, M. Arlitt, and C. Williamson, “A performance comparison of dynamic Web technologies,” ACM SIGMETRICS Performance Evaluation Review, 2003.

[13] J. Touch, J. Heidemann, and K. Obraczka, “Analysis of HTTP performance,” USC / Information Sciences Institute, 1998.

[14] Q. Wu, Y. Wang, “Performance Testing and Optimization of J2EE-Based Web Applications,” Education Technology and Computer Science (ETCS), 2010 Second International Workshop, Vol. 2, pp. 681-683, 2010.