메모리 덤프

jmap -dump:format=b,file=test.hprof 7757

vm상태

vmstat 1

http://d2.naver.com/helloworld/1326256

하나의 메모리 누수를 잡기까지

2015.06.04

|

41353

웹 서비스는 멀티 스레드 환경이기 때문에 항상 신중하게 코드를 작성해야 합니다. 부주의하게 수정한 소스 코드 때문에 메모리 누수가 일어나 애플리케이션이 응답하지 않는 503 오류가 발생한 경험이 있습니다. 이 글에서는 503 오류를 처리하고 그 원인을 찾아 메모리 누수 문제를 해결한 사례를 이야기하겠습니다.

장애 징조 - 상반기의 장애 처리 일지

잠자리에 들기 전에 늘 휴대폰을 챙긴다. 배터리가 충분한지, 매너 모드를 해제했는지 확인하고 손이 잘 닿는 위치에 휴대폰을 놓는다. 아침에 잠을 깨우는 알람이 울리기 전에 다른 연락이 없기를 바라며 잠이 든다. 서비스를 운영하다 보니 장애 관제 시스템에서 보내는 장애 메시지를 놓치지 않도록 신경을 곤두세운다. 잠을 푹 자는 건지는 모르겠다.

1월 4일 오후 11시 24분, 간헐적으로 503 오류가 발생한다는 알림을 받았다. 서비스에 투입한 장비 가운데 한 대에서 메모리 용량이 부족하다고 한다. 로그를 분석하니 메모리 부족(out of memory) 오류다. 로컬 장비에서 캐시를 운영하지만 힙 영역의 크기를 크게 설정하지 않은 것이 문제로 보인다. 힙 영역의 크기를 늘렸다.

4월 12일 오전 7시 47분, 또다시 간헐적으로 503 오류가 발생한다는 알림이다. 일찍 출근해 있었기에 Tomcat을 다시 시작해 우선 서비스가 동작하게 했다. 스레드 덤프(thread dump)를 저장하지 않아 로그만으로는 정확한 원인을 파악하기 힘들다. 다음부터는 스레드 덤프를 저장하기로 했다. 3개월 동안 아무 문제가 없었는데 원인이 무엇일까?

5월 11일 오후 8시 45분, 간헐적 503 오류다. 통합 로그 시스템에서 먼저 오류를 알리는 문자 메시지를 보내기 시작했다. 야근을 하던 개발자가 조치를 취했다. 이번에는 정확한 원인을 파악할 수 있지 않을까 기대했다. 로그 파일 분석으로는 데이터베이스의 커넥션 풀(connection pool)이 모자라는 곳에서 문제가 시작된 것으로 보인다. DBCP(database connection pool) 라이브러리의 문제라고 결론을 내렸다. 사내에서 수정한 commons-dbcp-1.2.2로 DBCP 라이브러리를 교체했다. 추가로 Apache Commons Pool 라이브러리도 1.4 버전으로 업그레이드했다. 다른 서비스에도 같은 문제가 있어 이번 기회에 DBCP에 관련된 문제를 전체 개발 조직에 공유했다.

모니터링 개선

나 혼자 죽을 수 없다

웹 서버에 문제가 생기면 L4 수준에서 자동으로 제외돼 여러 대의 서버를 사용해도 큰 문제가 없다. 하지만 Tomcat에 문제가 생겼을 때는 문제가 생긴 서비스가 L4 수준에서 자동으로 제외되지 않아 사용자 요청 중 일정 비율은 제대로 처리되지 않는 문제가 생긴다. 그래서 서비스를 출시할 때부터 Tomcat은 별도의 URL로 모니터링하게 만들어 Tomcat에 문제가 생기면 모니터링 시스템에서 확인할 수 있다. 장애 관리 시스템의 기록을 보면 응답 모니터링 시스템에서 장애가 확인되었다고 하니까 모니터링이 잘 된다고 보면 될 것 같다(그 이후로 L7 수준에서 장애를 확인하도록 해 Tomcat이 정지되면 자동으로 인프라 담당 부서에 문의해 서비스가 사용하는 L4 스위치에 여유가 있는 경우 L7 수준에서 서버를 전환한다. 또 다른 방법은 mod\_jk에서 로드 밸런서를 사용하는 방법이다.).

혹시 장애 관리 시스템에서 알리기 전에 서비스의 문제를 알 수 있는 방법은 없을까? 모니터링에서 실패를 조금이라도 먼저 감지하면 조치가 빠르지 않을까? 방법을 찾다 모니터링 시스템에서 개발자에게 문자 메시지를 보낼 수 있는 것을 알았다. 웹 서버나 Tomcat뿐만 아니라 CPU 사용량이 어느 정도 이상 올라가도 알려 줄 수 있다고 한다.

먼저 알고 먼저 처리할 수단이 있으면 마다할 이유가 없다. CPU 사용량까지 포함해 문제가 발생하면 개발자가 문자 메시지를 먼저 받게 인프라 담당 부서에 요청했다. 이때 우리 팀 개발자가 모든 팀원이 문자 메시지를 받도록 인프라 담당 부서에 요청했다. 마치 "(장애 담당자인) 나 혼자 죽을 수 없다"라고 마음 속으로 외치는 소리가 들리는 듯하다.

문자 메시지 스트레스

6월 16일부터 모니터링 개선의 효과를 보기 시작했다. 오후 10시 38분에 7번 기계의 CPU 사용량이 비정상적으로 높아진 것을 통합 모니터링 시스템에서 문자 메시지로 알려 주었다. TV를 보면서 맥주를 한잔하고 있었는데, 인프라 담당 부서에서 먼저 조치를 취하고 메일을 보내 주었다.

문자 메시지가 팀원 모두에게 가면 어떤 일이 생길까? 38명의 목격자가 있는 가운데 한 명이 살해당할 때 아무도 도움을 안 주지 않았다는 글을 읽은 적이 있다. 그 이유는 책임이 분산되기 때문이라고 한다. 즉, '모두 문자 메시지를 받으니 누군가 처리하겠지'라고 생각할 수 있는 것이다. 다행히도 우리 팀은 자발적인 팀이고 모두 책임 의식이 강하다. 우리는 거의 모든 팀원들이 문자 메시지를 받으면 회사의 시스템에 로그인한다.

6월21일 일요일 오후에 다시 문자 메시지가 왔다. 이번에는 8번 기계다. CPU 사용량이 올라갔다가 다시 안정화되었다. 그러나 다음날 새벽 1시에 다시 CPU 사용량이 100%까지 올라갔고 결국 해당 기계를 서비스에서 제외했다.

모니터링 개선 덕에 서비스에 문제가 생기기 전에 미리 서비스에 지장이 없도록 조치할 수 있었다. 그러나 계속해서 문자 메시지에 시달리면서 살 수는 없었다. 문자 메시지가 주는 스트레스도 만만치 않다.

문제 원인 찾기

이제 문제의 원인을 제거할 때다. 서비스에서 제외된 장비의 로그를 분석하기 시작했다. 스레드 덤프 분석 도구인 [Samurai](http://samuraism.jp/samurai)로 스레드 덤프를 분석했더니 교착 상태(deadlock)도 없고 모두 유휴 상태(idle)이거나 실행 중인 상태였다. 그리고 [PMAT(pattern modeling and analysis tool)](http://www.alphaworks.ibm.com/tech/pmat)로 가비지 컬렉션(garbage collection) 상태를 그래프로 그렸다.

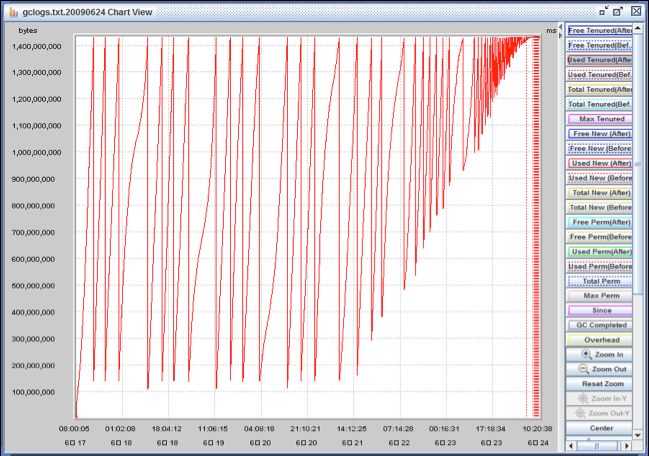


그림 1 가비지 컬렉션 상태 그래프

**PMAT와 -verbose 옵션** PMAT는 -verbose:gc 옵션으로 JVM을 실행할 때 출력되는 가비지 컬렉션 로그를 파싱해서 그래프로 보여 주는 도구다. 힙 메모리의 사용 패턴을 분석해서 적절한 설정값을 추천하는 기능도 제공한다. -verbose 옵션에 관해서는 "[Garbage Collection 모니터링 방법](http://d2.naver.com/helloworld/6043)"을 참고한다.

특정 시점 이후로는 가비지 컬렉션 이후에도 남는 메모리가 계속 줄어든다. 결국 메모리 누수가 원인이다. 몇 달에 한 번 발생하는 문제이기 때문에 개발 장비나 테스트 장비에서는 발생하지 않는다. 결국은 실제로 운영하는 서버에서 확인할 수밖에 없는데 어떤 서버에서 발생할지 모르고, 그렇다고 모든 서버에 프로파일러를 설치할 수는 없는 노릇이다. 메모리 누수가 발생하기 시작하는 장비를 찾아야 한다. 팀에서는 다음 메일과 같이 간단하면서도 효과적인 방법을 고안했다.

|  |
| --- |
| 아래처럼 웹 서버에서 메일을 발송하도록 스크립트를 만들었습니다. 보시면 대략 아시겠지만 01 ~ 12번 장비까지 최근 Full GC 발생 후 Old Generation의 추이고요.   현재 힙 메모리의 최대 용량은 1,398,144K까지 가능하고 가비지 컬렉션 후에는 대략 10% 이하가 유지되므로 아래 결과는 정상입니다.  혹시 아래 예처럼 가비지 컬렉션 후에도 최대 용량에 육박하는 장비가 발견된다면 조치가 필요한 것입니다.  [Full 1394849K -> 1350000K (1398144K)] [Full 1397258K -> 1351000K (1398144K)] [Full 1395333K -> 1352000K (1398144K)] [Full 1396319K -> 1353000K (1398144K)] [Full 1395599K -> 1354000K (1398144K)]  메모리 누수 상태에서도 이틀 이상은 유지되니깐 매일 오전 9시에 동작시키도록 할게요.  감사합니다. |

그리고 6월 26일에 드디어 꼬리를 잡았다.

|  |
| --- |
| 보낸 사람: 개발자 J   2번 장비 낌새가 좀 이상하군요.  [Full 1397073K -> 116835K (1398144K)] [Full 1395193K -> 142874K (1398144K)] [Full 1395081K -> 146435K (1398144K)] [Full 1395310K -> **374266K** (1398144K)] [Full 1395357K -> **629070K**(1398144K)] |

장비를 서비스에서 제외시키고 동작 중인 Tomcat으로 원인 분석을 시작했다.

문제 해결

히스토그램 비교

우선 힙 메모리의 히스토그램을 이용해 문제를 해결할 방법을 찾았다. 메모리 누수를 잡기 위해서 프로파일러를 사용하더라도 객체(object)의 변동을 추적하고 힙 영역의 내용을 볼 수밖에 없다. jmap으로 히스토그램을 받아서 정상 상태의 히스토그램과 비교해 증분이 많은 객체를 중점적으로 보면 원인을 찾을 수 있을 것이라고 가정했다. 객체의 개수와 크기를 비교해 CSV(comma separated values) 파일로 저장하는 프로그램을 만들었다. 실제 객체는 힙 덤프(heap dump)로 볼 생각이었다. 여기에 동원한 다음과 같은 도구는 모두 무료로 JDK 6에 포함돼 있다.

* jmap: 힙 덤프나 히스토그램을 출력하는 프로그램
* jhat: 힙 덤프를 이용해 각 객체를 볼 수 있는 프로그램
* VisualVM: 실행되고 있는 JVM의 힙 내용을 볼 수 있는 프로그램

다음은 히스토그램의 일부다.

$ jmap -histro:live 8825 | more

num # instances #bytes class name

----------------------------------------------

1: 3062256 677810312 [C

2: 3176949 76246776 java.lang.String

3: 29959 32072704 [I

4: 380080 27365760 xxx.xxx.common.model.xxxx

5: 100476 12792648 <constMethodKlass>

6: 113714 11254840 [Ljava.lang.Object;

7: 453459 11883016 java.util.HashMap$Entry

8: 100476 8043896 <methodKlass>

9: 16052 6917504 [B

10: 412877 6606032 java.lang.Integer

11: 141665 6546672 <symbolKlazss>

12: 10410 5282344 <const PoolKlass>

13: 12577 4451720 [Ljava.util.HashMap$Entry;

14: 10410 4351624 <instanceKlassKlass>

15: 9166 3134352 <constantPoolCacheKlass>

16: 105039 2520936 java.util.ArrayList

17: 58627 1876064 xx.xxx.xxx.xxxListEntry

18: 58027 1856864 java.util.LinkedHashMap$Entry

19: 21897 1751760 java.lang.reflect.Method

다음은 비교 결과를 Excel로 정리한 화면이다.

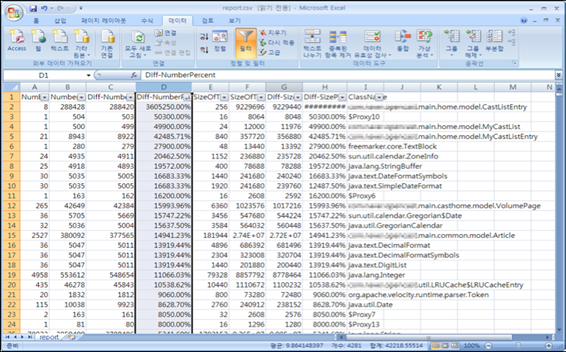


그림 2 힙 메모리의 히스토그램 비교

Excel로 결과를 보면 숫자 혹은 크기 변동에 따라 정렬할 수 있어서 문제를 찾기 쉬우리라 예상했다. 그러나 히스토그램을 비교하려면 어느 정도 시간 이상 실행되고 있는 서버의 히스토그램을 확보해야 하는데, 서비스 중인 서버에서 받지를 못했고 숫자만 봐서는 정확히 집어내기가 힘들었다. 어쩔 수 없이 히스토그램을 이용하는 방법은 다음 기회(다음 기회가 없기를 바란다)에 다시 시도하기로 했다.

힙 덤프 분석

히스토그램 비교 대신 문제가 발생한 서버의 힙 메모리 덤프를 받아서 분석을 시작했다. 힙 덤프 파일의 크기는 약 858MB 정도였는데, JDK에 포함된 jhat으로는 3시간이 지나도록 응답이 없어 분석을 진행할 수 없었다.

분석을 도와 줄 도구로 Eclipse의 [Memory Analyzer(MAT)](http://eclipse.org/mat)를 추천받아 사용했다. Memory Analyzer는 용량이 큰 힙 덤프 파일을 매우 빠른 속도로 탐색할 수 있고 Eclipse 플러그인 형태로도 제공된다. 실제로 Memory Analyzer를 사용했을 때 858MB의 힙 덤프 파일을 1분 이내로 읽어 분석을 진행할 수 있었다.

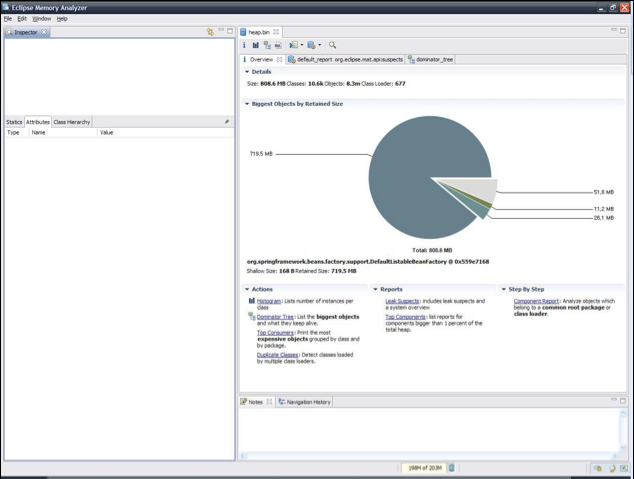


그림 3 Memory Analyzer의 힙 덤프 분석 결과

Memory Analyzer를 이용해서 가장 많은 메모리를 차지하고 있는 객체를 찾을 수 있었다. 약 717MB 정도 차지하고 있던 이 객체는 java.util.LinkedHashMap으로 만든 LRU(least recently used) 캐시였다. 캐시 엔트리 한 개의 크기는 약 20 ~ 40KB 정도고 최대 100개의 엔트리를 저장할 수 있다. 일정 시간이 지나거나 오랫동안 사용하지 않은 객체는 삭제하기 때문에 이론적으로는 40MB 이상의 메모리를 사용할 수 없었다.

LinkedHashMap 내부에 있는 table이라는 이름의 배열(array)에 데이터를 저장하는데, table[795]에 저장된 객체가 대부분의 용량을 차지하고 있음을 쉽게 알 수 있었다. 그래서 이 객체의 내부를 살펴보았더니 다음과 같았다.

LinkedHashMap은 LinkedList로 구현한 HashMap이며 맵에 저장한 엔트리를 before와 after라는 이름의 레퍼런스를 사용해 순서대로 엮는다. 따라서 before와 after 레퍼런스에 연결된 객체는 LinkedHashMap의 엔트리여야 하므로 내부의 table 배열에 존재하거나 다른 엔트리의 next 레퍼런스에 연결돼야 한다(next 레퍼런스는 해시의 버킷 연결을 의미한다). 그러나 table[795]에 저장된 객체의 before 레퍼런스에 연결된 객체는 table 배열에 존재하지 않았고 다른 엔트리의 next 레퍼런스에 연결되지도 않았다. 이것은 이 객체가 HashMap의 엔트리가 아님을 의미한다.

HashMap의 엔트리가 아닌 객체는 HashMap에서 지워야 하는데 어떠한 이유로 before 레퍼런스를 정리하지 못해 아직 남아 있기 때문에 가비지 컬렉터(garbage collector)가 객체를 정리하지 못한 상황이라고 볼 수 있었다. 그럼 왜 before 레퍼런스를 정리할 수 없었을까? 이런 문제는 대부분 멀티 스레드(multi thread) 환경에서 스레드 세이프(thread-safe)를 고려하지 않은 코드를 사용했을 때 발생한다는 것을 경험으로 알고 있었다. before 레퍼런스가 변경되는 시점의 코드를 찾아서 어떤 실수가 있었는지 확인하는 작업을 진행했다.

다음은 LinkedHashMap 클래스의 소스 코드 중 일부다.

/\*\*

\* Inserts this entry before the speicifed existing entry in the list.

\*/

private void addBefore(Entry<K,V> exsitingEntry){

after = existingEntry;

before = exsitingEntry.before;

before.after = this;

after.before = this;

}

/\*\*

\* This method is invoked by the superclass whenever the value

\* of a pre-existing entry is rady by Map.get or modifed by Map.set.

\* If the enclosing Map is access-ordered, it moves the entry

\* to the end of the list; otherwise, ti does nothing.

\*/

void recordAccess(HashMap<K,V> m) {

LinkedHashMap<K,V> lm = (LinkedHashMap <K,V>) m;

if (lm.accessOrder) {

lm.modCount++;

remove();

addBefore(lm.header);

}

}

....

LinkedHashMap 클래스 내부에서 before 레퍼런스를 변경할 때는 addBefore()라는 메서드를 호출하며, addBefore() 메서드는 recordAccess() 메서드에서 호출한다. recordAccess() 메서드는 LinkedHashMap.get() 메서드와 LinkedHashMap.put() 메서드에서 사용한다. 따라서, get() 메서드와 put() 메서드를 호출하는 부분은 스레드 세이프를 고려해 작성해야 한다.

Collection 객체를 다룰 때 put() 메서드를 사용할 때는 스레드 세이프를 고려해 syncronized 블록을 사용하지만, get() 메서드를 다룰 때는 생략하는 경우가 있다. 위 코드를 확인하는 순간 가장 먼저 든 생각은 '내가 get() 메서드'를 사용할 때 synchronized 블록을 사용했던가?'였다. 작성한 코드를 찾아보니 다음과 같았다.

...

public V get(K key) {

LRUCacheEntry<V> entry = null;

V result = null;

if (key !=null ) {

entry = cache.get(key);

if ( entry != null ) {

if (lifetime > 0 && (System.currentTimeMills() - entry.getEntranceTime() > lifetime )

synchronized (cache) {

cache.remove(key);

}

} else {

result = entry.getEntry();

}

}

}

return result;

}

우려했던 대로 LinkedHashMap.get() 메서드를 호출할 때 syncronized 블록을 사용하지 않았다. 따라서 메모리 누수의 원인을 다음과 같이 추정할 수 있었다.

1. 여러 스레드가 동시에 get() 메서드를 호출한다.
2. LinkedHashMap 내부에서 before 레퍼런스와 after 레퍼런스를 변경할 때 레퍼런스가 올바르게 연결되지 않는다.
3. 오동작으로 의도하지 않은 레퍼런스가 남은 객체는 가비지 컬렉터가 처리할 수 없으므로 메모리를 계속 차지하게 된다.
4. 이와 같은 일이 반복되면 차지하는 메모리가 점점 증가하게 되고 결국 메모리 부족 오류가 발생한다.

문제의 코드를 스레드 세이프를 고려해 수정한 다음 서비스에 반영했다. 이후 지속적으로 메모리 상황을 관찰한 결과 메모리 누수 현상은 완전히 사라졌다.

마치며

가장 어이없었던 일은 이 다음에 일어났다. LRUCache 클래스는 전부터 사용하던 오래된 코드였기 때문에 이러한 버그가 있다는 사실을 현재 담당 부서에 알리고 수정한 코드를 보내야 했다. 하지만 서둘러 예전 소스를 체크아웃하고 해당 클래스의 소스 코드를 확인하는 순간 뒤통수를 한 대 얻어맞은 기분이었다. 예전 코드는 스레드 세이프를 고려해 synchronized 블록을 사용해서 작성돼 있었다!

정리하자면 이렇다. 예전에 작성한 코드를 재사용하면서 자세히 알아보지도 않고 '이거 왜 get() 메서드를 synchronized 블록으로 감싸지? 속도만 느려지잖아? 누군가 실력이 모자라 이렇게 했군. 아우 창피해' 뭐 이런 생각으로 가볍게 수정했던 것이다. 웹 서비스는 멀티 스레드 환경이기 때문에 항상 신중하게 코드를 작성해야 하는데 가볍게 생각한 부분이 큰 문제를 불러 일으켰다. 개인적으로는 좋은 경험이었지만 팀원들과 관련 부서, 서비스를 이용하는 고객에게 불편을 끼쳐 대단히 죄송스러웠다. 속된 말로 좀 많이 '쪽팔렸다'.

오랜 시간이 지났지만 이때를 생각하면 여전히 창피하다. 대단할 것 없는 작은 사례지만 부디 동료들이 '쪽팔리는' 경험을 줄이는 데 도움이 되었으면 하는 마음을 담아서 이 고해성사 같은 글을 마무리한다.