Helion

Charlie Hunt, Binu John

Wydajność JAVY



Poznaj i wykorzystaj
optymalne sposoby na regulowanie
wydajności oprogramowania Java!

Tytuł oryginału: Java Performance on Multi-Core Platforms, First Edition

Tłumaczenie: Lech Lachowski

Projekt okładki: proj: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

ISBN: 978-83-246-4380-6

Authorized translation from the English language edition, entitled: JAVA PERFORMANCE ON MULTI-CORE PLATFORMS, First Edition; ISBN 0137142528; by Charlie Hunt; and John Binu; published by Pearson Education, Inc, publishing as Prentice Hall.
Copyright © 2012 by Oracle and/or its affiliates. All rights reserved. Oracle is headquartered at 500 Oracle Parkway, Redwood Shores, CA 94065, U.S.A.

All rights reserved. No part of this book may by reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.
Polish language edition published by HELION S.A. Copyright © 2013.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Wydawnictwo HELION dołożyło wszelkich starań, by zawarte w tej książce informacje były kompletne
i rzetelne. Nie bierze jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Wydawnictwo HELION nie ponosi również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/wydjav.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/wydjav
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Przedmowa **13**

Przedmowa **15**

Wstęp **17**

Podziękowania **21**

O autorach **23**

1. Strategie, podejścia i metodologie **25**
Zasadnicze czynniki 26
Dwa podejścia: z góry do dołu oraz z dołu do góry 29
Metoda z góry do dołu 29
Metoda z dołu do góry 30
Wybór odpowiedniej platformy i ocena systemu 31
Wybór właściwej architektury procesora 32
Ocena wydajności systemu 33
Bibliografia 34

2. Monitorowanie wydajności systemu operacyjnego **35**
Definicje 36
Wykorzystanie CPU 36
Monitorowanie wykorzystania CPU w systemach Windows 37
Monitorowanie wykorzystania CPU w systemach Windows za pomocą komendy typeperf 40
Monitorowanie wykorzystania CPU w systemach Linux 41
Monitorowanie wykorzystania CPU w systemach Solaris 42
Monitorowanie wykorzystania CPU w systemach Linux i Solaris za pomocą narzędzi wiersza poleceń 45

Kup książkę Poleć książkę

Kolejka uruchamiania planisty krótkoterminowego ... 48
Monitorowanie kolejki uruchamiania planisty krótkoterminowego w systemach Windows ... 49
Monitorowanie kolejki uruchamiania planisty krótkoterminowego w systemach Solaris ... 50
Monitorowanie kolejki uruchamiania planisty krótkoterminowego w systemach Linux ... 51
Wykorzystanie pamięci ... 51
Monitorowanie wykorzystania pamięci w systemach Windows ... 52
Monitorowanie wykorzystania pamięci w systemach Solaris ... 53
Monitorowanie wykorzystania pamięci w systemach Linux ... 54
Monitorowanie rywalizacji o blokady w systemach Solaris ... 55
Monitorowanie rywalizacji o blokady w systemach Linux ... 57
Monitorowanie rywalizacji o blokady w systemach Windows ... 57
Izolowanie gorących blokad ... 58
Monitorowanie mimowolnego przełączania kontekstu ... 58
Monitorowanie migracji wątków ... 59
Wykorzystanie we/wy sieci ... 59
Monitorowanie wykorzystania we/wy sieci w systemach Solaris ... 60
Monitorowanie wykorzystania we/wy sieci w systemach Linux ... 61
Monitorowanie wykorzystania we/wy sieci w systemach Windows ... 61
Względy dotyczące poprawy wydajności aplikacji ... 62
Wykorzystanie we/wy dysku ... 63
Dodatkowe narzędzia wiersza poleceń ... 65
Monitorowanie wykorzystania CPU w systemach z procesorami SPARC T ... 66
Bibliografia ... 69

3. Przegląd JVM ... 71
Wysokopoziomowa architektura HotSpot VM ... 72
HotSpot VM Runtime ... 73
Opcje wiersza poleceń ... 74
Cykl życia maszyny wirtualnej ... 75
Ładowanie klas maszyny wirtualnej ... 78
Weryfikacja kodu bajtowego ... 80
Udostępnianie danych klas ... 81
Interpreter ... 82
Obsługa wyjątków ... 84
Synchronizacja ... 84
Zarządzanie wątkami ... 85
Zarządzanie stertą C++ ... 89
Java Native Interface ... 90
Obsługa błędów krytycznych VM ... 91
Mechanizmy odzyskiwania pamięci HotSpot VM ... 93
Pokoleniowy mechanizm odzyskiwania pamięci ... 93
Młode pokolenie ... 95
Szybka alokacja ... 97
Mechanizmy odzyskiwania pamięci, czyli osiołkowi w żłoby dano ... 97

Kup książkę Poleć książkę

Szeregowy mechanizm odzyskiwania pamięci ... 98
Równoległy mechanizm odzyskiwania pamięci: przepustowość ma znaczenie! ... 99
Przeważnie-równoczesny mechanizm odzyskiwania pamięci: opóźnienie ma znaczenie! ... 99
Mechanizm odzyskiwania pamięci najpierw-kosz: następca CMS ... 102
Porównanie ... 102
Generowanie pracy mechanizmu odzyskiwania pamięci ... 103
Perspektywa historyczna ... 103
Kompilatory JIT HotSpot VM ... 104
Analiza hierarchii klas ... 105
Polityka kompilacji ... 106
Deoptymalizacja ... 107
Kompilator JIT dla aplikacji klienckich ... 108
Kompilator JIT dla aplikacji serwerowych ... 108
SSA — wykres programowo zależny ... 108
Nadchodzące poprawki ... 110
Regulacja adaptacyjna HotSpot VM ... 111
Wartości domyślne dla Java 1.4.2 ... 111
Ergonomiczne wartości domyślne dla 5. wersji Javy ... 111
Zaktualizowane domyślne wartości ergonomiczne dla Java 6 Update 18 ... 113
Adaptacyjne ustalanie rozmiaru sterty Java ... 115
Nie tylko ergonomia ... 115
Bibliografia ... 116

4. Monitorowanie wydajności maszyny wirtualnej Javy ... 117
Definicje ... 118
Odzyskiwanie pamięci ... 118
Dane istotne dla procesu odzyskiwania pamięci ... 119
Raportowanie danych procesu odzyskiwania pamięci ... 119
Analiza offline danych z procesów odzyskiwania pamięci ... 129
Narzędzia graficzne ... 132
Kompilator JIT ... 150
Ładowanie klas ... 151
Monitorowanie aplikacji Java ... 153
Szybkie monitorowanie rywalizacji o blokady ... 154
Bibliografia ... 156

5. Profilowanie aplikacji Java ... 157
Terminologia ... 159
Terminy związane z profilowaniem ... 159
Pojęcia związane z programem Oracle Solaris Studio Performance Analyzer ... 159
Pojęcia związane z programem NetBeans Profiler ... 160
Oracle Solaris Studio Performance Analyzer ... 161
Obsługiwane platformy ... 161
Pobieranie i instalacja programu Oracle Solaris Studio Performance Analyzer ... 162
Przechwytywanie profilu Oracle Solaris Studio Performance Analyzer ... 163
Przeglądanie zebranej próby ... 166
Prezentacja danych ... 174

Kup książkę Poleć książkę

Filtrowanie danych profilu ... 177
Narzędzie wiersza poleceń er_print ... 178
Program NetBeans Profiler ... 185
Obsługiwane platformy ... 186
Pobieranie i instalacja NetBeans Profiler ... 186
Rozpoczynanie sesji profilowania metod ... 186
Controls, czyli przyciski sterujące ... 193
Status ... 193
Profiling Results, czyli rezultaty profilowania ... 193
Saved Snapshots, czyli zapisane zrzuty ekranu ... 193
View, czyli podgląd ... 194
Basic Telemetry, czyli podstawowa telemetria ... 194
Przeglądanie aktualnych rezultatów ... 195
Wykonywanie zrzutu ekranu rezultatów ... 195
Rozpoczynanie sesji profilowania pamięci ... 196
Przeglądanie bieżących rezultatów ... 198
Wykonywanie zrzutu ekranu rezultatów ... 200
Izolowanie wycieków pamięci ... 201
Analiza zrzutów sterty ... 201
Bibliografia ... 202

6. Profilowanie aplikacji Java — porady i sztuczki ... 203
Potencjalne obszary poprawy wydajności ... 203
Wykorzystanie CPU przez jądro lub system ... 204
Rywalizacja o blokady ... 212
Użycie słowa kluczowego volatile ... 221
Zmiana rozmiaru struktur danych ... 222
Zmiana rozmiaru klas StringBuilder lub StringBuffer ... 223
Zmiana rozmiaru klas Java Collections ... 226
Zwiększanie współczynnika równoległości ... 230
Wysokie wykorzystanie CPU ... 232
Inne użyteczne wskazówki dotyczące programu Performance Analyzer ... 233
Bibliografia ... 235

7. Regulacja JVM krok po kroku ... 237
Metodologia ... 238
Założenia ... 240
Wymagania dotyczące infrastruktury testowania ... 240
Wymagania systemowe aplikacji ... 241
Dostępność ... 241
Zarządzalność ... 241
Przepustowość ... 241
Opóźnienie i czas reakcji ... 242
Zużycie pamięci ... 242
Czas uruchamiania ... 242
Ranking wymagań systemowych ... 242
Wybór modelu wdrożenia JVM ... 243
Model wdrożenia pojedynczej instancji JVM ... 243

Kup książkę Poleć książkę

Model wdrożenia wielu instancji JVM ... 243
Wskazówki ogólne ... 244
Wybór środowiska uruchomieniowego JVM ... 244
Środowisko uruchomieniowe typu klienckiego czy serwerowego ... 244
32-bitowa czy 64-bitowa maszyna wirtualna Javy ... 245
Mechanizmy odzyskiwania pamięci ... 246
Podstawy regulowania mechanizmu odzyskiwania pamięci ... 246
Atrybuty wydajności ... 246
Podstawowe zasady ... 247
Opcje wiersza poleceń oraz rejestrowanie zdarzeń dla mechanizmu odzyskiwania pamięci ... 247
Określanie zużycia pamięci ... 251
Ograniczenia ... 251
Układ sterty HotSpot VM ... 252
Punkt początkowy rozmiaru sterty ... 254
Obliczanie rozmiaru żywych danych ... 256
Konfiguracja początkowego rozmiaru sterty ... 257
Dodatkowe uwagi ... 259
Regulowanie opóźnienia/czasu reakcji ... 259
Przesłanki ... 260
Precyzyjna regulacja rozmiaru przestrzeni młodego pokolenia ... 261
Precyzyjna regulacja rozmiaru przestrzeni starego pokolenia ... 264
Precyzyjna regulacja opóźnień dla równoczesnego mechanizmu odzyskiwania pamięci ... 267
Objaśnienie przestrzeni ocalałych ... 269
Objaśnienie progu zatrudnienia ... 271
Monitorowanie progu zatrudnienia ... 272
Ustalanie rozmiaru przestrzeni ocalałych ... 274
Bezpośrednio wywoływane procesy odzyskiwania pamięci ... 280
Równoczesne odzyskiwanie pamięci dla stałego pokolenia ... 281
Regulacja czasu przestojów CMS ... 282
Kolejne czynności ... 283
Regulowanie przepustowości aplikacji ... 283
Regulacja przepustowości dla CMS ... 284
Regulacja przepustowościowego mechanizmu odzyskiwania pamięci ... 285
Regulacja rozmiaru przestrzeni ocalałych ... 287
Regulacja liczby wątków przepustowościowego mechanizmu odzyskiwania pamięci ... 290
Wdrożenie na systemach NUMA ... 291
Kolejne czynności ... 291
Przypadki skrajne ... 291
Dodatkowe opcje wiersza poleceń do regulowania wydajności ... 292
Najnowsze i największe optymalizacje ... 292
Analiza ucieczki ... 292
Blokada przeciągana ... 293
Duże strony pamięci ... 294
Bibliografia ... 296

Kup książkę Poleć książkę

8. Testy porównawcze aplikacji Java 297
Wyzwania dotyczące benchmarków 298
Rozgrzewka 298
Odzyskiwanie pamięci 300
Zastosowanie metod Time interfejsów Java API 301
Usuwanie martwego kodu w wyniku optymalizacji 302
Wplatanie 307
Deoptymalizacja 311
Porady dotyczące przygotowywania mikrobenchmarków 315
Projektowanie eksperymentów 317
Zastosowanie metod statystycznych 318
Obliczanie średniej 318
Obliczanie odchylenia standardowego 319
Określanie przedziału ufności 319
Zastosowanie testów hipotezy 321
Porady dotyczące stosowania metod statystycznych 323
Literatura 324
Bibliografia 324

9. Testy porównawcze aplikacji wielowarstwowych 325
Wyzwania dotyczące benchmarków 326
Rozważania na temat benchmarków typu enterprise 328
Definiowanie systemu testowego (SUT) 328
Przygotowywanie mikrobenchmarków 329
Definiowanie modelu interakcji z użytkownikiem 329
Definiowanie metryk wydajności 333
Skalowanie benchmarku 337
Weryfikacja za pomocą prawa Little'a 338
Czas na zastanowienie 340
Analiza skalowalności 343
Przeprowadzanie benchmarku 343
Monitorowanie serwera aplikacji 347
Monitorowanie za pomocą serwera aplikacji GlassFish 347
Monitorowanie podsystemów 352
Wydajność systemów zewnętrznych 356
We/wy dysku 359
Monitorowanie i regulacja pul zasobów 361
Profilowanie aplikacji enterprise 362
Bibliografia 363

10. Wydajność aplikacji internetowych 365
Testy porównawcze aplikacji internetowych 366
Komponenty kontenera webowego 367
Konektor HTTP 368
Silnik serwletu 369
Monitorowanie i regulowanie wydajności kontenera webowego 369
Tryb deweloperski i tryb produkcyjny kontenera 370
Menedżer bezpieczeństwa 371

Kup książkę Poleć książkę

Regulacje JVM ... 371
HTTP service i kontener webowy ... 373
Nasłuchiwacz HTTP ... 373
Najlepsze praktyki ... 386
Najlepsze praktyki dla serwletów i stron JSP ... 386
Buforowanie pamięci podręcznej zawartości ... 395
Trwałość sesji ... 400
Buforowanie plików serwera HTTP ... 401
Bibliografia ... 405

11. Wydajność usług internetowych ... 407
Wydajność XML ... 408
Cykl przetwarzania XML ... 408
Parsowanie/unmarshalling ... 409
Dostęp ... 412
Modyfikacja ... 412
Serializacja/marshalling ... 413
Weryfikacja ... 413
Rozwiązywanie encji zewnętrznych ... 415
Częściowe przetwarzanie dokumentów XML ... 417
Wybór odpowiedniego interfejsu API ... 420
Stos modelowej implementacji JAX-WS ... 423
Testy porównawcze usług internetowych ... 425
Czynniki, które wpływają na wydajność usługi internetowej ... 428
Wpływ rozmiaru wiadomości ... 428
Charakterystyki wydajności różnych typów schematów ... 430
Implementacja punktu końcowego ... 433
Wydajność programu obsługi ... 434
Najlepsze praktyki dotyczące wydajności ... 436
Przetwarzanie binarnego bloku danych ... 436
Praca z dokumentami XML ... 441
Wykorzystanie MTOM do wysyłania dokumentów XML w postaci załączników ... 441
Korzystanie z interfejsu provider ... 444
Fast Infoset ... 446
Kompresja HTTP ... 448
Wydajność klienta usługi internetowej ... 449
Bibliografia ... 450

12. Wydajność Java Persistence oraz Enterprise Java Beans ... 451
Model programowania EJB ... 452
Interfejs Java Persistence API i jego implementacja modelowa ... 453
Pamięć podręczna drugiego poziomu ... 453
Monitorowanie i regulacja kontenera EJB ... 456
Pula wątków ... 457
Pule beanów i pamięci podręczne beanów ... 459
Pamięć podręczna sesji EclipseLink ... 464
Poziom izolacji transakcji ... 465

Kup książkę Poleć książkę

Najlepsze praktyki stosowane w Enterprise Java Beans 466
Benchmark do testów porównawczych EJB wykorzystany w przykładach 466
EJB 2.1 466
EJB 3.0 478
Najlepsze praktyki stosowane w Java Persistence 481
Zapytania języka zapytań JPA 482
Pamięć podręczna wyników zapytań 484
FetchType 485
Tworzenie puli połączeń 486
Aktualizacje zbiorcze 488
Wybierz właściwą strategię blokowania bazy danych 489
Odczyty bez transakcji 490
Dziedziczenie 490
Bibliografia 490

A. Wybrane opcje wiersza poleceń HotSpot VM 493

Skorowidz 511

Kup książkę Poleć książkę

4

Monitorowanie wydajności maszyny wirtualnej Javy

W tym rozdziale opisane zostały elementy, które powinny być monitorowane na poziomie stosu programowego maszyny wirtualnej Javy (ang. *Java Virtual Machine* — JVM). Ponadto przedstawione zostały narzędzia służące do monitorowania JVM oraz najbardziej typowe, powtarzające się wzorce, na które należy zwracać uwagę. Szczegóły dotyczące podejmowania na podstawie zebranych informacji decyzji dotyczących regulacji JVM znajdziesz w rozdziale 7. Na końcu rozdziału zamieszczony został również krótki podrozdział na temat monitorowania aplikacji.

Monitorowanie maszyny wirtualnej Javy jest czynnością, która zawsze powinna być wykonywana z aplikacją typu produkcyjnego. Ponieważ JVM jest kluczowym elementem stosu programowego, powinna być monitorowana w tym samym stopniu, co sama aplikacja oraz system operacyjny. Analiza informacji uzyskanych w procesie monitorowania maszyny wirtualnej Javy pozwala stwierdzić, kiedy niezbędna jest regulacja JVM. Taka regulacja maszyny wirtualnej Javy powinna być przeprowadzana za każdym razem, kiedy zmienia się wersja JVM, dochodzi do zmian w systemie operacyjnym (zmiana konfiguracji lub wersji), zmienia się wersja aplikacji lub wykonywane są aktualizacje oraz gdy dochodzi do znaczących zmian w zakresie danych wejściowych dla aplikacji. Do zmiany danych wejściowych aplikacji może dochodzić dość często w wielu aplikacjach Java, co może mieć wpływ na wydajność JVM. Dlatego też monitorowanie maszyny wirtualnej Javy jest tak istotną czynnością.

Istnieje kilka podstawowych obszarów JVM, które powinny być monitorowane. Należą do nich procesy odzyskiwania pamięci, działanie kompilatora JIT oraz ładowanie klas. Dostępnych jest wiele narzędzi umożliwiających monitorowanie maszyny wirtualnej Javy. Niektóre z nich są dystrybuowane z pakietami JDK, inne są darmowe, a jeszcze inne to wersje komercyjne (płatne).

Kup książkę Poleć książkę

Opisane w tym rozdziale narzędzia do monitorowania należą do grupy dystrybuowanych z pakietami Oracle JDK, do grupy narzędzi darmowych lub też do aplikacji typu open source. Ponadto wszystkie prezentowane tu narzędzia dostępne są dla systemów operacyjnych Windows, Linux oraz Oracle Solaris (zwanego dalej Solaris).

Aby właściwie zrozumieć materiał przedstawiony w tym rozdziale, pomocna będzie wiedza na temat najważniejszych komponentów oraz głównych operacji nowoczesnej maszyny wirtualnej Javy. Informacje na temat Java HotSpot VM i jej komponentów zostały przedstawione w rozdziale 3.

Definicje

Zanim zagłębimy się w szczegóły dotyczące elementów, które należy monitorować, przydatne może okazać się przypomnienie pojęć „monitorowanie wydajności" oraz „profilowanie wydajności" wprowadzonych na początku rozdziału 2. **Monitorowanie wydajności** (ang. *performance monitoring*) polega na nieinwazyjnym zbieraniu i obserwowaniu danych dotyczących wydajności działającej lub uruchomionej aplikacji. Jest to zazwyczaj działanie typu prewencyjnego lub zapobiegawczego, może więc być przeprowadzane w środowisku produkcyjnym, kwalifikacyjnym lub programistycznym. Monitorowanie wydajności jest również pierwszym etapem reagowania w sytuacji, kiedy interesariusze danej aplikacji zgłaszają występowanie problemów z wydajnością, a nie zapewniają przy tym wystarczających informacji lub wskazówek na temat potencjalnej przyczyny takiego stanu rzeczy. W takim przypadku prawdopodobne jest, że po fazie monitorowania wydajności nastąpi faza profilowania wydajności. Monitorowanie wydajności pozwala również zidentyfikować lub wyizolować potencjalne problemy bez istotnego wpływania na czasy reakcji oraz przepustowość aplikacji.

Z kolei **profilowanie wydajności** (ang. *performance profiling*) jest czynnością zbierania danych dotyczących wydajności działającej lub uruchomionej aplikacji, która to czynność może wpływać na przepustowość lub reakcje aplikacji. Profilowanie wydajności jest z reguły działaniem reakcyjnym lub działaniem, które ma stanowić odpowiedź na problemy z wydajnością zgłaszane przez zainteresowane osoby. Koncentruje się ono zazwyczaj na węższym obszarze niż monitorowanie wydajności. Profilowanie jest rzadko przeprowadzane w środowiskach produkcyjnych. Najczęściej wykonywane jest w środowisku kwalifikującym, testującym lub programistycznym i nierzadko następuje po monitorowaniu wydajności.

W odróżnieniu od monitorowania wydajności i profilowania wydajności, **regulacja wydajności** (ang. *performance tuning*) polega na zmianie wartości parametrów regulowanych, kodu źródłowego lub atrybutów konfiguracji w celu poprawy przepustowości i reakcji. Regulacja wydajności często następuje po monitorowaniu lub profilowaniu wydajności.

Odzyskiwanie pamięci

Monitorowanie procesów odzyskiwania pamięci JVM jest istotnym czynnikiem, ponieważ może mieć głęboki wpływ na przepustowość i opóźnienia aplikacji. Nowoczesne maszyny wirtualne Javy, takie jak Java HotSpot VM (zwana dalej HotSpot VM), zapewniają możliwość obserwacji statystyk odzyskiwania pamięci dla każdego procesu odzyskiwania. Statystyki dostarczane są w formie tekstowej, zapisywane bezpośrednio do pliku dziennika (ang. *log file*) lub publikowane w interfejsie graficznym (GUI) systemu monitorującego.

Podrozdział ten rozpoczyna się od przedstawienia listy danych, które są istotne dla procesu odzyskiwania pamięci. Następnie zaprezentowany został zestaw opcji wiersza poleceń HotSpot VM

Kup książkę Poleć książkę

służący do raportowania statystyk odzyskiwania pamięci. Zamieszczono przy tym objaśnienie znaczenia raportowanych danych. Dodatkowo przedstawione zostało narzędzie graficzne, które można wykorzystać do analizy uzyskanych danych. I co najważniejsze, znajduje się tutaj również opis konkretnych wzorców czy układów danych, na które należy zwracać uwagę, oraz sugestie dotyczące tego, kiedy trzeba wykonać regulację mechanizmu odzyskiwania pamięci JVM.

Dane istotne dla procesu odzyskiwania pamięci

W statystykach procesu odzyskiwania pamięci istotne są następujące dane:

- stosowany mechanizm odzyskiwania pamięci;
- rozmiar sterty Java;
- rozmiary przestrzeni młodego i starego pokolenia;
- rozmiar przestrzeni stałego pokolenia;
- czasy trwania procesów mniejszego odzyskiwania pamięci;
- częstotliwość uruchamiania procesu mniejszego odzyskiwania pamięci;
- ilość przestrzeni odzyskiwanej w procesach mniejszego odzyskiwania pamięci;
- czasy trwania procesów pełnego odzyskiwania pamięci;
- częstotliwość uruchamiania procesu pełnego odzyskiwania pamięci;
- ilość przestrzeni odzyskiwanej w cyklach równoczesnego odzyskiwania pamięci;
- zapełnienie sterty Java przed procesem odzyskiwania pamięci i po nim;
- zapełnienie przestrzeni młodego i starego pokolenia przed procesem odzyskiwania pamięci i po nim;
- zapełnienie przestrzeni stałego pokolenia przed procesem odzyskiwania pamięci i po nim;
- określenie, kiedy proces pełnego odzyskiwania pamięci wywoływany jest przez zapełnienie przestrzeni starego pokolenia, a kiedy przez zapełnienie przestrzeni stałego pokolenia;
- ustalenie, kiedy aplikacja korzysta z wyraźnych wywołań metody `System.gc()`.

Raportowanie danych procesu odzyskiwania pamięci

Raportowanie danych statystycznych dla procesów odzyskiwania pamięci nie stanowi dużego obciążenia dla maszyny wirtualnej HotSpot. Obciążenie to jest w rzeczywistości tak znikome, że zaleca się gromadzenie danych odzyskiwania nawet w środowisku produkcyjnym. W tym punkcie opisano kilka różnych opcji wiersza poleceń HotSpot VM służących do generowania statystyk działania mechanizmu odzyskiwania pamięci oraz objaśniono znaczenie tych statystyk.

Zasadniczo istnieją dwa typy odzyskiwania pamięci: mniejsze odzyskiwanie pamięci (ang. *minor garbage collection*), zwane również odzyskiwaniem pamięci z młodego pokolenia (ang. *young generation garbage collection*), oraz pełne odzyskiwanie pamięci (ang. *full garbage collection*), zwane również głównym odzyskiwaniem pamięci (ang. *major garbage collection*). Ogólnie rzecz biorąc, pełne odzyskiwanie pamięci obejmuje, oprócz usuwania obiektów „śmieciowych", także kompaktowanie przestrzeni starego i stałego pokolenia. Istnieje jednak kilka wyjątków od tej reguły. W maszynie wirtualnej HotSpot domyślnym zachowaniem przy pełnym odzyskiwaniu pamięci jest usuwanie śmieci z przestrzeni młodego, starego i stałego pokolenia. Ponadto w tym procesie kompaktowane są przestrzenie starego i stałego pokolenia, a wszystkie żywe obiekty z młodego pokolenia są promowane do starego pokolenia. Dlatego też na koniec procesu pełnego odzyski-

Kup książkę Poleć książkę

wania pamięci przestrzeń młodego pokolenia jest pusta, a przestrzenie starego i młodego pokolenia są skompaktowane i przechowują jedynie żywe obiekty. Sposoby działania każdego z mechanizmów odzyskiwania pamięci maszyny wirtualnej HotSpot zostały szczegółowo opisane w rozdziale 3.

Jak już wcześniej wspominaliśmy, mniejsze odzyskiwanie pamięci uwalnia obszary pamięci zajmowane przez nieosiągalne obiekty znajdujące się w przestrzeni młodego pokolenia. Z kolei domyślnym zachowaniem dla pełnego odzyskiwania pamięci w HotSpot VM jest uwalnianie obszarów pamięci zajmowanych przez nieosiągalne obiekty z przestrzeni młodego, starego i stałego pokolenia. Można skonfigurować maszynę wirtualną HotSpot tak, aby przy pełnym odzyskiwaniu pamięci obiekty „śmieciowe" z przestrzeni młodego pokolenia nie były zbierane, zanim nie zostaną zebrane obiekty „śmieciowe" z przestrzeni starego pokolenia. Służy do tego opcja wiersza poleceń `-XX:-ScavengeBeforeFullGC`. Znak `-` umieszczony przed `ScavengeBeforeFullGC` oznacza wyłączenie odzyskiwania pamięci w młodym pokoleniu dla procesu pełnego odzyskiwania pamięci. Z kolei zastosowanie znaku + przed tą opcją oznaczałoby włączenie odzyskiwania pamięci w przestrzeni młodego pokolenia dla procesu pełnego odzyskiwania pamięci. Jak już pisaliśmy, domyślnym zachowaniem dla HotSpot VM jest włączenie odzyskiwania w młodym pokoleniu dla pełnego odzyskiwania pamięci. Zalecane jest stosowanie ustawień domyślnych i niewyłączanie przestrzeni młodego pokolenia z procesu pełnego odzyskiwania. Zbieranie obiektów „śmieciowych" w przestrzeni młodego pokolenia przed wykonaniem tej samej czynności dla przestrzeni starego pokolenia zazwyczaj przysparza mniej pracy mechanizmowi odzyskiwania pamięci oraz powoduje, że więcej obiektów „śmieciowych" jest usuwanych, ponieważ obiekty w przestrzeni starego pokolenia mogą posiadać referencje do obiektów w przestrzeni młodego pokolenia. Jeśli nie zostanie przeprowadzone odzyskiwanie pamięci dla przestrzeni młodego pokolenia, żaden obiekt starego pokolenia, który posiada referencje do obiektów młodego pokolenia, nie może być usunięty.

-XX:+PrintGCDetails

Mimo iż `-verbose:gc` jest najczęściej stosowaną opcją wiersza poleceń dla raportowania statystyk procesu odzyskiwania pamięci, opcja `-XX:+PrintGCDetails` dostarcza dodatkowych i bardziej istotnych informacji w tym zakresie. W tej podsekcji zaprezentowano przykładowy listing z `-XX:+PrintGC ↪Details` dla przepustowościowego oraz równoczesnego mechanizmu odzyskiwania pamięci wraz z wyjaśnieniem dostarczonych danych. Opisane są również pewne wzorce, których należy szukać w listingu tej opcji.

Warto zwrócić uwagę, że dodatkowe informacje dostarczane przez opcję `-XX:+PrintGCDetails` mogą różnić się w zależności od wersji maszyny wirtualnej HotSpot.

Poniżej znajduje się przykład listingu dla opcji `-XX:+PrintGCDetails` przy przepustowościowym mechanizmie odzyskiwania pamięci w Java 6 Update 25. Przepustowościowy mechanizm odzyskiwania pamięci może być włączony za pomocą opcji `-XX:+UseParallelGC` lub `-XX:+Use ↪ParallelOldGC`. Układ listingu został podzielony na kilka wierszy w celu uzyskania lepszej przejrzystości.

```
[GC
    [PSYoungGen: 99952K->14688K(109312K)]
    422212K->341136K(764672K), 0.0631991 secs]
    [Times: user=0.83 sys=0.00, real=0.06 secs]
```

Oznaczenie `GC` wskazuje na proces mniejszego odzyskiwania pamięci. Fragment `[PSYoungGen: 99952K>14688K(109312K)]` dostarcza informacji na temat przestrzeni młodego pokolenia. `PSYoungGen` oznacza, że stosowane odzyskiwanie dla młodego pokolenia jest wielowątkowym procesem przepustowościowego mechanizmu odzyskiwania pamięci włączonego opcją wiersza poleceń `-XX:+Use ↪ParallelGC` lub automatycznie uruchomionego przez `-XX:+UseParallelOldGC`. Inne dostępne

Kup książkę Poleć książkę

mechanizmy odzyskiwania pamięci dla młodego pokolenia to: `ParNew`, czyli wielowątkowy proces odzyskiwania pamięci dla młodego pokolenia stosowany przez równoczesny mechanizm odzyskiwania pamięci starego pokolenia znany jako CMS, oraz `DefNew`, czyli jednowątkowy proces odzyskiwania pamięci dla młodego pokolenia stosowany przez szeregowy mechanizm odzyskiwania pamięci, który jest uruchamiany opcją wiersza poleceń `-XX:+UseSerialGC`. Opcja `-XX:+UseSerialGC` (`DefNew`) może być również stosowana w połączeniu z równoczesnym mechanizmem odzyskiwania pamięci starego pokolenia (CMS), co zapewnia jednowątkowe odzyskiwanie pamięci dla młodego pokolenia. Obecnie mechanizm odzyskiwania pamięci G1, nad którym nadal trwają prace, nie stosuje identyfikatora do identyfikacji danych wyjściowych w taki sposób, jak pozostałe trzy mechanizmy odzyskiwania pamięci.

Wartość `99952K` znajdująca się po lewej stronie strzałki `->` to zapełnienie przestrzeni młodego pokolenia przed uruchomieniem procesu odzyskiwania pamięci. Wartość `14688K` po prawej stronie strzałki `->` to zapełnienie przestrzeni młodego pokolenia po przeprowadzeniu procesu odzyskiwania pamięci. Przestrzeń młodego pokolenia podzielona jest na przestrzeń zwaną edenem oraz dwie przestrzenie ocalałych. Ponieważ po mniejszym odzyskiwaniu pamięci eden jest zawsze pusty, `14688K` oznacza zapełnienie przestrzeni ocalałych. Wartość podana w nawiasach, czyli `(109312K)`, to całkowity rozmiar (nie mylić z zapełnieniem) przestrzeni młodego pokolenia, na którą składa się eden oraz dwie przestrzenie ocalałych.

W następnym wierszu listingu znajdują się wartości `422212K->341136K(764672K)`, które dostarczają informacji na temat wykorzystania sterty Java (łączna zajętość przestrzeni młodego i starego pokolenia) przed odzyskiwaniem pamięci i po nim. Ponadto znajduje się tu informacja o rozmiarze sterty Java, który jest równy sumie przestrzeni młodego i starego pokolenia. Wartość `422212K` po lewej stronie strzałki to zajętość sterty Java przed uruchomieniem procesu odzyskiwania pamięci, a wartość `341136K` po prawej stronie strzałki to zajętość sterty po przeprowadzeniu procesu odzyskiwania pamięci. Ujęta w nawiasach wartość `(764672K)` to całkowity rozmiar sterty Java.

Korzystając z uzyskanych informacji na temat rozmiaru przestrzeni młodego pokolenia oraz rozmiaru sterty Java, możesz obliczyć rozmiar przestrzeni starego pokolenia. Jeśli przykładowo rozmiar sterty Java wynosi `764672K`, a rozmiar przestrzeni młodego pokolenia jest równy `109312K`, to rozmiar przestrzeni starego pokolenia wynosi `764672K - 109312K = 655360K`.

Wartość `0.0631991 secs` określa w sekundach czas trwania procesu odzyskiwania pamięci.

Kolejny wiersz listingu `[Times: user=0.06 sys=0.00, real=0.06 secs]` to informacja na temat wykorzystania CPU oraz czasu, jaki upłynął. Wartość dla pola `user` to czas CPU wykorzystany przez mechanizm odzyskiwania pamięci na wykonanie instrukcji poza systemem operacyjnym. W tym przykładzie mechanizm odzyskiwania pamięci wykorzystał 0,06 s czasu użytkownika dla CPU. Wartość dla pola `sys` to czas CPU wykorzystany przez system operacyjny na obsługę mechanizmu odzyskiwania pamięci. W naszym przykładzie mechanizm odzyskiwania pamięci w ogóle nie wykorzystał czasu CPU na wykonanie instrukcji systemu operacyjnego. Wartość dla pola `real` to podany w sekundach czas, jaki upłynął od momentu rozpoczęcia do momentu zakończenia procesu odzyskiwania pamięci. W tym przypadku zajęło to 0,06 s. Czasy dla `user`, `sys` i `real` są zaokrąglane do setnych sekundy.

W poniższym przykładzie przedstawiono listing z opcji `-XX:+PrintGCDetails` dla procesu pełnego odzyskiwania pamięci. Dla lepszej przejrzystości listing został podzielony na kilka wierszy.

```
[Full GC
    [PSYoungGen: 11456K->0K(110400K)]
    [PSOldGen: 651536K->58466K(655360K)]
    662992K->58466K(765760K)
    [PSPermGen: 10191K->10191K(22528K)],
    1.1178951 secs]
    [Times: user=1.01 sys=0.00, real=1.12 secs]
```

Kup książkę Poleć książkę

Nagłówek `Full GC` wskazuje, że mamy do czynienia z pełnym odzyskiwaniem pamięci. `[PSYoung ↪Gen: 11456K->0K(110400K)]` oznacza to samo, co w przypadku opisanego wcześniej mniejszego odzyskiwania pamięci.

Wiersz `[PSOldGen: 651536K->58466K(655360K)]` dostarcza informacji na temat przestrzeni starego pokolenia. `PSOldGen` oznacza, że stosowany mechanizm odzyskiwania pamięci dla starego pokolenia jest wielowątkowym procesem przepustowościowego mechanizmu odzyskiwania pamięci włączonego opcją wiersza poleceń `XX:+UseParallelOldGC`. Wartość `651536K` po lewej stronie strzałki to zapełnienie przestrzeni starego pokolenia przed uruchomieniem odzyskiwania pamięci, a wartość `58466K` po prawej stronie strzałki to zapełnienie tej przestrzeni po zakończeniu procesu odzyskiwania pamięci. Ujęta w nawiasach wartość `(655360K)` to rozmiar przestrzeni starego pokolenia.

Wiersz `662992K->58466K(765760K)` informuje o wykorzystaniu sterty Java. Jest tu podane łączne zapełnienie przestrzeni młodego i starego pokolenia przed wykonaniem i po wykonaniu odzyskiwania pamięci. Wartość po prawej stronie strzałki może być również traktowana jako ilość żywych obiektów, które pozostały w aplikacji po przeprowadzeniu pełnego odzyskiwania pamięci. Posiadanie informacji o ilości żywych obiektów w aplikacji, szczególnie kiedy aplikacja jest w stanie równowagi, jest istotne przy ustalaniu rozmiaru sterty Java maszyny wirtualnej oraz dostosowywaniu ustawień mechanizmu odzyskiwania pamięci w JVM.

Kolejny wiersz `[PSPermGen: 10191K->10191K(22528K)]` zawiera dane dotyczące przestrzeni stałego pokolenia. `PSPermGen` oznacza, że stosowany mechanizm odzyskiwania pamięci dla stałego pokolenia jest wielowątkowym procesem przepustowościowego mechanizmu odzyskiwania pamięci uruchomionego opcją wiersza poleceń `-XX:+UseParallelGC` lub `-XX:+UseParallelOldGC`. Wartość `10191K` znajdująca się po lewej stronie strzałki określa zapełnienie przestrzeni stałego pokolenia przed uruchomieniem procesu odzyskiwania pamięci, a wartość `10191K` znajdująca się po prawej stronie strzałki to zapełnienie tej przestrzeni po zakończeniu pracy przez mechanizm odzyskiwania pamięci. Ujęta w nawiasach wartość `(22528K)` to rozmiar przestrzeni stałego pokolenia.

W pełnym odzyskiwaniu pamięci istotne jest, by zwrócić uwagę na to, jaka część stery jest zajęta przez przestrzenie starego i stałego pokolenia przed uruchomieniem mechanizmu odzyskiwania pamięci. Powodem tego jest fakt, że pełne odzyskiwanie pamięci może być uruchamiane w przypadku zbliżania się limitu zapełnienia zarówno dla przestrzeni starego, jak i stałego pokolenia. Z naszego listingu wynika, że zapełnienie przestrzeni starego pokolenia przed odzyskiwaniem pamięci (`651536K`) jest bliskie całkowitemu rozmiarowi tej przestrzeni (`655360K`). Z kolei zapełnienie przestrzeni stałego pokolenia przed odzyskiwaniem pamięci (`10191K`) nie jest nawet blisko całkowitego rozmiaru tej przestrzeni (`22528K`). Stąd też wiemy, że uruchomienie pełnego odzyskiwania pamięci zostało spowodowane osiągnięciem limitu zapełnienia przestrzeni starego pokolenia.

Wartość `1.1178951 secs` określa w sekundach czas trwania procesu odzyskiwania pamięci.

Wiersz `[Times: user=1.01 sys=0.00, real=1.12 secs]` dostarcza informacji na temat wykorzystania CPU i czasu, jaki upłynął. Jego znaczenie jest takie samo, jak w przypadku opisanego wcześniej procesu mniejszego odzyskiwania pamięci.

Przy zastosowaniu równoczesnego mechanizmu odzyskiwania pamięci (CMS) listing dla opcji `-XX:+PrintGCDetails` nieco się różni, szczególnie w przypadku raportowania danych dotyczących przeważnie-równoczesnego odzyskiwania pamięci dla starego pokolenia. Równoczesny mechanizm odzyskiwania pamięci (CMS) uruchamiany jest opcją wiersza poleceń `-XX:+UseConcMarkSweepGC`. Uruchamia on również automatycznie opcję `-XX:+UseParNewGC`, czyli wielowątkowe odzyskiwanie pamięci dla młodego pokolenia. Poniżej znajduje się przykład dla mniejszego odzyskiwania pamięci wykorzystującego równoczesny mechanizm odzyskiwania pamięci CMS:

Kup książkę Poleć książkę

```
[GC
    [ParNew: 2112K->64K(2112K), 0.0837052 secs]
    16103K->15476K(773376K), 0.0838519 secs]
    [Times: user=0.02 sys=0.00, real=0.08 secs]
```

Listing z mniejszego odzyskiwania pamięci dla równoczesnego mechanizmu odzyskiwania pamięci jest zbliżony do listingu z mniejszego odzyskiwania pamięci dla przepustowościowego mechanizmu odzyskiwania pamięci. Aby zapewnić kompleksowość informacji, został on opisany poniżej.

Nagłówek `GC` wskazuje, że mamy do czynienia z mniejszym odzyskiwaniem pamięci. Wiersz `[ParNew: 2112K->64K(2112K)]` zawiera informacje na temat przestrzeni młodego pokolenia. `ParNew` wskazuje, że stosowane odzyskiwanie pamięci dla młodego pokolenia jest wielowątkowym procesem równoczesnego mechanizmu odzyskiwania pamięci CMS. Jeśli zostałoby zdefiniowane użycie z CMS szeregowego odzyskiwania pamięci dla młodego pokolenia, mielibyśmy oznaczenie `DefNew`.

Wartość `2112K` znajdująca się po lewej stronie strzałki określa zapełnienie przestrzeni młodego pokolenia przed uruchomieniem mechanizmu odzyskiwania pamięci, a wartość `64K` po prawej stronie strzałki oznacza zapełnienie tej przestrzeni po zakończeniu działania mechanizmu odzyskiwania pamięci. Przestrzeń młodego pokolenia dzieli się na eden oraz dwie przestrzenie ocalałych. Ponieważ eden zawsze pozostaje pusty po mniejszym odzyskiwaniu pamięci, `64K` oznacza zapełnienie przestrzeni ocalałych. Ujęta w nawiasach wartość `(2112K)` jest rozmiarem przestrzeni młodego pokolenia, na którą składa się eden i dwie przestrzenie ocalałych. `0.0837052 secs` to wyrażony w sekundach czas, jaki był potrzebny na usunięcie nieosiągalnych obiektów z przestrzeni młodego pokolenia.

Kolejny wiersz listingu — `16103K->15476K(773376K)` — przedstawia dane na temat wykorzystania sterty Java (łączne zapełnienie przestrzenie młodego i starego pokolenia) przed wykonaniem i po wykonaniu procesu odzyskiwania pamięci. Ponadto dostarcza również informacji dotyczącej rozmiaru sterty Java, która jest równa sumie rozmiarów przestrzeni młodego i starego pokolenia. Wartość `16103K` znajdująca się po lewej stronie strzałki oznacza zapełnienie sterty Java przed uruchomieniem mechanizmu odzyskiwania pamięci, a wartość `15476K` po prawej stronie strzałki oznacza zapełnienie sterty po zakończeniu działania mechanizmu odzyskiwania pamięci. Ujęta w nawiasach wartość `(773376K)` stanowi całkowity rozmiar sterty Java.

Korzystając z uzyskanych informacji na temat rozmiaru przestrzeni młodego pokolenia oraz rozmiaru sterty Java, możesz obliczyć rozmiar przestrzeni starego pokolenia. Jeśli przykładowo rozmiar sterty Java wynosi `773376K`, a rozmiar przestrzeni młodego pokolenia jest równy `2112K`, to rozmiar przestrzeni starego pokolenia wynosi `773376K - 2112K = 771264K`.

`0.0838519 secs` oznacza wyrażony w sekundach czas trwania procesu mniejszego odzyskiwania pamięci, włącznie z usuwaniem „śmieciowych" obiektów z przestrzeni młodego pokolenia i promowaniem obiektów do przestrzeni starego pokolenia oraz końcowymi procesami oczyszczania.

W wierszu `[Times: user=0.02 sys=0.00, real=0.08 secs]` znajdują się informacje na temat wykorzystania CPU oraz czasu, jaki upłynął. Wartość dla pola `user` to czas CPU wykorzystany przez mechanizm odzyskiwania pamięci na wykonanie instrukcji poza systemem operacyjnym. W tym przykładzie mechanizm odzyskiwania pamięci wykorzystał 0,02 s czasu użytkownika dla CPU. Wartość dla pola `sys` to czas CPU wykorzystany przez system operacyjny na obsługę mechanizmu odzyskiwania pamięci. W naszym przykładzie mechanizm odzyskiwania pamięci w ogóle nie wykorzystał czasu CPU na wykonanie instrukcji systemu operacyjnego. Wartość dla pola `real` to podany w sekundach czas, jaki upłynął od momentu rozpoczęcia do momentu zakończenia procesu odzyskiwania pamięci. W tym przypadku zajęło to 0,08 s. Czasy dla `user`, `sys` i `real` są zaokrąglane do setnych sekundy.

Jak pewnie wiesz z opisu mechanizmu CMS zamieszczonego w rozdziale 3., istnieje cykl przeważnie-równoczesnego odzyskiwania pamięci, który może być wykorzystywany w starym

Kup książkę Poleć książkę

pokoleniu. Opcja `-XX:+PrintGCDetails` raportuje również aktywność odzyskiwania pamięci dla każdego cyklu równoczesnego odzyskiwania pamięci. W poniższym przykładzie przedstawiono listing z odzyskiwania pamięci, który raportuje cały cykl równoczesnego odzyskiwania pamięci. Aktywność równoczesnego odzyskiwania pamięci jest przeplatana z procesami mniejszego odzyskiwania pamięci, aby wykazać, że procesy mniejszego odzyskiwania pamięci mogą wystąpić podczas cyklu równoczesnego odzyskiwania pamięci. Dla uzyskania lepszej przejrzystości dane dotyczące równoczesnego odzyskiwania pamięci zostały oznaczone wytłuszczonym drukiem. Pamiętaj również, że listing z opcji `-XX:+PrintGCDetails` dla mechanizmu CMS może różnić się w zależności od wersji JVM.

```
[GC
    [1 CMS-initial-mark: 13991K(773376K)]
    14103K(773376K), 0.0023781 secs]
    [Times: user=0.00 sys=0.00, real=0.00 secs]
[CMS-concurrent-mark-start]
[GC
    [ParNew: 2077K->63K(2112K), 0.0126205 secs]
    17552K->15855K(773376K), 0.0127482 secs]
    [Times: user=0.01 sys=0.00, real=0.01 secs]
[CMS-concurrent-mark: 0.267/0.374 secs]
    [Times: user=4.72 sys=0.01, real=0.37 secs]
[GC
    [ParNew: 2111K->64K(2112K), 0.0190851 secs]
    17903K->16154K(773376K), 0.0191903 secs]
    [Times: user=0.01 sys=0.00, real=0.02 secs]
[CMS-concurrent-preclean-start]
[CMS-concurrent-preclean: 0.044/0.064 secs]
    [Times: user=0.11 sys=0.00, real=0.06 secs]
[CMS-concurrent-abortable-preclean-start]
[CMS-concurrent-abortable-clean]  0.031/0.044 secs]
    [Times: user=0.09 sys=0.00, real=0.04 secs]
[GC
    [YG occupancy: 1515 K (2112K)
    [Rescan (parallel) , 0.0108373 secs]
    [weak refs processing, 0.0000186 secs]
    [1 CMS-remark: 16090K(20288K)]
    17242K(773376K), 0.0210460 secs]
    [Times: user=0.01 sys=0.00, real=0.02 secs]
[GC
     [ParNew: 2112K->63K(2112K), 0.0716116 secs]
     18177K->17382K(773376K), 0.0718204 secs]
     [Times: user=0.02 sys=0.00, real=0.07 secs]
[CMS-concurrent-sweep-start]
[GC
     [ParNew: 2111K->63K(2112K), 0.0830392 secs]
     19363K->18757K(773376K), 0.0832943 secs]
     [Times: user=0.02 sys=0.00, real=0.08 secs]
[GC
     [ParNew: 2111K->0K(2112K), 0.0035190 secs]
     17527K->15479K(773376K), 0.0036052 secs]
     [Times: user=0.00 sys=0.00, real=0.00 secs]
[CMS-concurrent-sweep: 0.291/0.662 secs]
   [Times: user=0.28 sys=0.01, real=0.66 secs]
[GC
     [ParNew: 2048K->0K(2112K), 0.0013347 secs]
     17527K->15479K(773376K), 0.0014231 secs]
     [Times: user=0.00 sys=0.00, real=0.00 secs]
[CMS-concurrent-reset-start]
```

Kup książkę Poleć książkę

```
[CMS-concurrent-reset: 0.016/0.016 secs]
     [Times: user=0.01 sys=0.00, real=0.02 secs]
[GC
     [ParNew: 2048K->1K(2112K), 0.0013936 secs]
     17527K->15479K(773376K), 0.0014814 secs]
     [Times: user=0.00 sys=0.00, real=0.00 secs]
```

Cykl CMS rozpoczyna się przestojem dla fazy znaku początkującego (ang. *initial mark*) i kończy po wykonaniu fazy równoczesnego resetu (ang. *concurrent reset*). W powyższym listingu każda z faz cyklu CMS została zaznaczona wytłuszczonym drukiem i rozpoczyna się od `CMS-initial-mark`, a kończy na `CMS-concurrent-reset`. Pozycja `CMS-concurrent-mark` oznacza zakończenie fazy równoczesnego znakowania. Z kolei pozycja `CMS-concurrent-sweep` wskazuje na zakończenie fazy równoczesnego zamiatania. Pozycje `CMS-concurrent-preclean` oraz `CMS-concurrent-abortable-preclean` określają zadania, które mogą być wykonywane równocześnie w ramach przygotowań do fazy ponownego zaznaczania (ang. *remark phase*) oznaczonej jako `CMS-remark`. Faza zamiatania oznaczona `CMS-concurrent-sweep` jest fazą, w której uwalniana jest przestrzeń zajmowana przez obiekty zaznaczone jako nieosiągalne. Końcową fazę wskazuje pozycja `CMS-concurrent-reset`. Jest to faza przygotowania do rozpoczęcia kolejnego cyklu równoczesnego mechanizmu odzyskiwania pamięci.

Znak inicjujący to z reguły relatywnie krótki przestój w stosunku do czasu, jaki jest potrzebny na wykonanie procesu mniejszego odzyskiwania pamięci. Czas, jaki jest niezbędny na wykonanie faz równoczesnych (równoczesnego zaznaczania, równoczesnego czyszczenia wstępnego oraz równoczesnego zamiatania), może być relatywnie długi (tak jak w powyższym przykładzie) w porównaniu do przestoju dla mniejszego odzyskiwania pamięci, jednak w trakcie wykonywania faz równoczesnych wątki aplikacji Java nie są zatrzymywane. Długość przestoju dla fazy ponownego zaznaczania jest zależna od charakterystyki aplikacji (np. wysoki współczynnik modyfikacji obiektów może wydłużać czas przestoju) oraz od czasu, jaki upłynął od momentu przeprowadzenia mniejszego odzyskiwania pamięci (duża liczba obiektów w przestrzeni młodego pokolenia może wydłużyć przestój).

Wzorcem, na który powinieneś zwrócić uwagę w tym listingu, jest stopień redukcji zapełnienia przestrzeni starego pokolenia podczas cyklu CMS. Przyjrzeć się powinieneś szczególnie temu, w jaki sposób spada zapełnienie sterty Java pomiędzy rozpoczęciem a zakończeniem fazy równoczesnego zamiatania CMS, co w listingu oznaczone zostało odpowiednio pozycjami `CMS-concurrent-sweep-start` oraz `CMS-concurrent-sweep`. Zapełnienie sterty Java możesz obserwować, śledząc procesy mniejszego odzyskiwania pamięci. Dlatego też powinieneś zwracać uwagę na mniejsze odzyskiwania pamięci pomiędzy rozpoczęciem a zakończeniem fazy równoczesnego zamiatania CMS. Jeśli w tym przedziale następuje niewielki spadek zapełnienia sterty Java, to albo zostało usuniętych niewiele obiektów (co oznacza, że cykle odzyskiwania pamięci CMS odnajdują niewiele nieosiągalnych obiektów do usunięcia, więc marnują wykorzystanie CPU), albo liczba obiektów promowanych do starego pokolenia jest równa lub wyższa niż współczynnik, z jakim faza równoczesnego zamiatania CMS jest w stanie je usuwać. Każdy z tych dwóch powyższych przypadków stanowi istotne wskazanie, że powinna zostać przeprowadzona regulacja maszyny wirtualnej Javy. Więcej informacji dotyczących regulacji mechanizmu odzyskiwania pamięci CMS znajdziesz w rozdziale 7.

Kolejnym artefaktem, który należy monitorować przy korzystaniu z CMS, jest dystrybucja zatrudnienia obiektów (ang. *tenuring distribution*) włączana opcją wiersza poleceń `-XX:+Print` `↪TenuringDistribution`. Dystrybucja zatrudnienia to histogram przedstawiający wiek obiektów w przestrzeniach ocalałych młodego pokolenia. Kiedy wiek obiektu przekroczy określony próg zatrudnienia (ang. *tenuring threshold*), jest promowany z młodego do starego pokolenia. Próg zatrudnienia oraz sposób monitorowania dystrybucji zatrudnienia wraz ze wskazówkami dotyczą-

Kup książkę Poleć książkę

cymi elementów, na które należy zwracać uwagę, zostały opisane w punktach „Objaśnienie progu zatrudnienia" i „Monitorowanie progu zatrudnienia", w rozdziale 7.

Jeśli obiekty są zbyt szybko promowane do starego pokolenia, a CMS nie jest w stanie zapewnić odpowiedniej ilości wolnej przestrzeni w stosunku do liczby obiektów promowanych z młodego do starego pokolenia, prowadzi to do wyczerpania dostępnej przestrzeni starego pokolenia. Sytuacja taka określana jest jako **niewydolność trybu równoczesnego** (ang. *concurrent mode failure*). Niewydolność trybu równoczesnego może również wystąpić, kiedy przestrzeń starego pokolenia zostanie pofragmentowana do tego stopnia, że nie będzie w niej już odpowiednio dużego miejsca na przyjęcie obiektu promowanego z przestrzeni młodego pokolenia. Opcja `-XX:+PrintGCDetails` raportuje niewydolność trybu równoczesnego w listingu z odzyskiwania pamięci, podając informację tekstową `(concurrent mode failure)`. Kiedy występuje niewydolność trybu równoczesnego, przeprowadzane jest odzyskiwanie pamięci w przestrzeni starego pokolenia w celu odzyskania wolnej przestrzeni. Przeprowadzane jest również kompaktowanie starego pokolenia, aby wyeliminować fragmentację. Cała ta operacja wymaga zatrzymania wątków aplikacji Java i może wymagać dość dużej ilości czasu. Dlatego też, jeśli zaobserwujesz powtarzające się przypadki niewydolności trybu równoczesnego, powinieneś wykonać regulację JVM według wskazówek zamieszczonych w rozdziale 7., szczególnie tych, które dotyczą dostrajania aplikacji w celu osiągnięcia małych opóźnień.

Znaczniki daty i czasu

Maszyna wirtualna HotSpot posiada opcje wiersza poleceń umożliwiające dołączenie znaczników daty i czasu w każdym raporcie dotyczącym odzyskiwania pamięci. Opcja wiersza poleceń `-XX:+Print`
`↪GCTimeStamps` drukuje znacznik czasu będący liczbą sekund, które upłynęły od momentu uruchomienia JVM. Znacznik ten zamieszczany jest dla każdego procesu odzyskiwania pamięci. W poniższym przykładzie przedstawiono listing dla mniejszego odzyskiwania pamięci z opcji `-XX:+Print`
`↪GCTimeStamps` w połączeniu z opcją `-XX:+PrintGCDetails` i przepustowościowym mechanizmem odzyskiwania pamięci. Listing został podzielony na kilka wierszy w celu uzyskania lepszej przejrzystości.

```
77.233: [GC
    [PSYoungGen: 99952K->14688K(109312K)]
    422212K->341136K(764672K), 0.0631991 secs]
    [Times: user=0.83 sys=0.00, real=0.06 secs]
```

Zwróć uwagę, że listing dla opcji `-XX:+PrintGCDetails` jest poprzedzony znacznikiem czasu reprezentującym liczbę sekund, które upłynęły od momentu uruchomienia JVM. Listing dla pełnego odzyskiwania pamięci również zawiera prefiks ze znacznikiem czasu. Znacznik ten jest dodawany także w przypadku stosowania równoczesnego mechanizmu odzyskiwania pamięci.

Od czasu ukazania się Java 6 Update 4 dostępna jest opcja wiersza poleceń `-XX:+PrintGCDate`
`↪Stamps`. Wstawia ona znacznik daty i czasu zgodny z normą ISO 860. Data i czas podawane są w formie *YYYY*`-`*MM*`-`*DD*`-T-`*HH*`-`*MM*`-`*SS*`.`*mmm*`-TZ`, gdzie:

- *YYYY* to czterocyfrowy format roku;
- *MM* to dwucyfrowe określenie miesiąca, wartości jednocyfrowe poprzedzane są cyfrą `0`;
- *DD* to dwucyfrowe określenie dnia miesiąca, wartości jednocyfrowe poprzedzane są cyfrą `0`;
- `T` jest znacznikiem informującym, że po jego lewej stronie znajduje się data, a po prawej określenie czasu dnia;
- *HH* to dwucyfrowe określenie godziny, wartości jednocyfrowe poprzedzane są cyfrą `0`;
- *MM* to dwucyfrowe określenie minuty, wartości jednocyfrowe poprzedzane są cyfrą `0`;

Kup książkę Poleć książkę

- *SS* to dwucyfrowe określenie sekundy, wartości jednocyfrowe poprzedzane są cyfrą 0;
- *mmm* to trzycyfrowe określenie milisekundy, wartości jednocyfrowe i dwucyfrowe poprzedzane są odpowiednio cyframi 00 oraz 0;
- TZ oznacza przesunięcie strefy czasowej względem czasu uniwersalnego GMT.

Mimo że listing zawiera informacje na temat przesunięcia strefy czasowej względem czasu GMT, to data i czas dnia nie są podawane według GMT, tylko dostosowywane do czasu lokalnego. Poniższy listing wykorzystuje opcję -XX:+PrintGCDateStamps wraz z opcją -XX:+PrintGCDetails przy zastosowaniu przepustowościowego mechanizmu odzyskiwania pamięci. Listing został podzielony na kilka wierszy w celu uzyskania lepszej przejrzystości.

```
2010-11-21T09:57:10.518-0500:[GC
    [PSYoungGen: 99952K->14688K(109312K)]
    422212K->341136K(764672K), 0.0631991 secs]
    [Times: user=0.83 sys=0.00, real=0.06 secs]
```

Przy pełnym odzyskiwaniu pamięci dla przepustowościowego mechanizmu odzyskiwania również dostępny jest przy zastosowaniu opcji -XX:+PrintGCDateStamps prefiks zawierający znacznik daty i czasu. Ponadto znaczniki daty i czasu drukowane są także przy korzystaniu z równoczesnego mechanizmu odzyskiwania pamięci.

Użycie znacznika daty i (lub) czasu umożliwia zmierzenie czasu trwania zarówno procesu mniejszego, jak i pełnego odzyskiwania pamięci łącznie z częstotliwością uruchamiania tych procesów. Na podstawie daty i czasu możesz obliczyć oczekiwaną częstotliwość przeprowadzania mniejszego i pełnego odzyskiwania pamięci. Jeśli czasy trwania lub częstotliwości uruchamiania procesów odzyskiwania pamięci przekraczają dopuszczalne normy dla aplikacji, powinieneś rozważyć regulację JVM według wskazówek zamieszczonych w rozdziale 7.

-Xloggc

Opcja wiersza poleceń HotSpot VM -Xloggc:*<nazwa_pliku>* ułatwia przeprowadzenie analizy offline statystyk odzyskiwania pamięci dzięki bezpośredniemu zapisaniu danych z raportu do pliku. Parametr *<nazwa_pliku>* pozwala zdefiniować dowolną nazwę dla pliku, w którym chcesz przechowywać dane dotyczące procesów odzyskiwania pamięci. Analiza offline tych danych może dotyczyć szerokiego przedziału czasu, co umożliwia identyfikację wzorców kształtowania się określonych statystyk bez konieczności obserwacji danych na bieżąco przy uruchomionej aplikacji.

Przy zastosowaniu opcji -XX:+PrintGCDetails w kombinacji z -Xloggc:*<nazwa_pliku>* listing jest automatycznie poprzedzany znacznikiem czasu nawet bez konieczności określania opcji -XX:+PrintGCTimeStamps. Znacznik czasu jest drukowany w ten sam sposób jak w przypadku -XX:+PrintGCTimeStamps. Poniżej znajduje się przykład dla opcji -Xloggc:*<nazwa_pliku>* w połączeniu z opcją -XX:+PrintGCDetails dla przepustowościowego mechanizmu odzyskiwania pamięci. Listing został podzielony na kilka wierszy w celu uzyskania lepszej przejrzystości.

```
77.233: [GC
    [PSYoungGen: 99952K->14688K(109312K)]
    422212K->341136K(764672K), 0.0631991 secs]
    [Times: user=0.83 sys=0.00, real=0.06 secs]
```

Ponieważ opcja -Xloggc dołącza do listingu znacznik czasu automatycznie, łatwo można określić, kiedy przeprowadzane są procesy mniejszego i pełnego odzyskiwania. Ponadto możesz również obliczyć częstotliwość uruchamiania tych procesów. Na podstawie daty i czasu możesz obliczyć oczekiwaną częstotliwość przeprowadzania mniejszego i pełnego odzyskiwania pamięci. Jeśli czasy trwania lub częstotliwości uruchamiania procesów odzyskiwania pamięci przekraczają dopusz-

Kup książkę Poleć książkę

czalne normy dla aplikacji, powinieneś rozważyć regulację JVM według wskazówek zamieszczonych w rozdziale 7.

Czas zatrzymania i czas równoczesny aplikacji

HotSpot VM może dostarczać informacji na temat ilości czasu, przez jaki aplikacja działa pomiędzy operacjami punktów bezpieczeństwa, oraz ilości czasu, przez jaki HotSpot VM blokuje wykonywanie wątków Java. Służą do tego opcje wiersza poleceń `-XX:+PrintGCApplicationConcurrentTime` oraz `-XX:+PrintGCApplicationStoppedTime`. Obserwacja operacji punktów bezpieczeństwa za pomocą tych dwóch opcji może dostarczyć informacji, które mogą być użyteczne dla zrozumienia i ilościowego określenia wpływu opóźnień generowanych przez JVM. Może to również pomóc w zdefiniowaniu, czy konkretne opóźnienie wynika z zachowania JVM spowodowanego operacją punktu bezpieczeństwa, czy jest związane z działaniem aplikacji.

Wskazówka

Operacje punktów bezpieczeństwa zostały opisane bardziej szczegółowo w rozdziale 3.

Poniżej znajduje się przykład zastosowania opcji `-XX:+PrintGCApplicationConcurrentTime` oraz `-XX:+PrintGCApplicationStoppedTime` dla `-XX:+PrintGCDetails`:

```
Application time: 0.5291524 seconds
[GC
    [ParNew: 3968K->64K(4032K), 0.0460948 secs]
    7451K->6186K(32704K), 0.0462350 secs]
    [Times: user=0.01 sys=0.00, real=0.05 secs]
Total time for which application threads were stopped: 0.0468229 seconds
Application time: 0.5279058 seconds
[GC
    [ParNew: 4032K->64K(4032K), 0.0447854 secs]
    10154K->8648K(32704K), 0.0449156 secs]
    [Times: user=0.01 sys=0.00, real=0.04 secs]
Total time for which application threads were stopped: 0.0453124 seconds
Application time: 0.9063706 seconds
[GC
    [ParNew: 4032K->64K(4032K), 0.0464574 secs]
    12616K->11187K(32704K), 0.0465921 secs]
    [Times: user=0.01 sys=0.00, real=0.05 secs]
Total time for which application threads were stopped: 0.0470484 seconds
```

Z listingu można wyczytać, że aplikacja działała w przybliżeniu od 0,53 do 0,91 sekundy z przestojami na mniejsze odzyskiwanie pamięci w przybliżeniu od 0,045 do 0,047. Stanowi to około 5 – 8% obciążenia na procesy mniejszego odzyskiwania pamięci.

Zwróć też uwagę, że nie ma żadnych dodatkowych punktów bezpieczeństwa pomiędzy każdym z procesów mniejszego odzyskiwania pamięci. Gdyby punkty bezpieczeństwa wystąpiły pomiędzy procesami odzyskiwania pamięci, w listingu pojawiłyby się następujące informacje dla każdego z tych punktów bezpieczeństwa: czas aplikacji (`Application time:`) oraz całkowity czas zatrzymania wątków aplikacji (`Total time for which application threads were stopped:`).

Wyraźne procesy odzyskiwania pamięci

Wyraźne procesy odzyskiwania pamięci mogą być łatwo zidentyfikowane na listingu. Listing z odzyskiwania pamięci zawiera informację tekstową wskazującą, że pełne odzyskiwanie pamięci jest

Kup książkę Poleć książkę

wynikiem wyraźnego wywołania metody `System.gc()`. W poniższym przykładzie przedstawiono listing z opcji wiersza poleceń `-XX:+PrintGCDetails` dla procesu pełnego odzyskiwania pamięci wywołanego metodą `System.gc()`. Listing został podzielony na kilka wierszy w celu uzyskania lepszej przejrzystości.

```
[Full GC (System)
    [PSYoungGen: 99608K->0K(114688K)]
    [PSOldGen: 317110K->191711K(655360K)]
    416718K->191711K(770048K)
    [PSPermGen: 15639K->15639K(22528K)],
    0.0279619 secs]
    [Times: user=0.02 sys=0.00, real=0.02 secs]
```

Zwróć uwagę, że pozycja `(System)` poprzedzona została oznaczeniem `Full GC`. Wskazuje to, że mamy do czynienia z procesem pełnego odzyskiwania pamięci wywołanym przez `System.gc()`. Jeśli odnajdziesz wyraźny proces odzyskiwania pamięci w logach odzyskiwania pamięci, powinieneś odszukać przyczynę jego zastosowania, a następnie zdecydować, czy wywołanie `System.gc()` powinno być usunięte z kodu źródłowego, czy też powinno zostać wyłączone.

Opcje wiersza poleceń polecane do monitorowania procesów odzyskiwania pamięci

Podstawowy zestaw opcji wiersza poleceń HotSpot VM służący do monitorowania procesów odzyskiwania pamięci to `-XX:+PrintGCDetails` wraz z opcją `-XX:+PrintGCTimeStamps` lub `-XX:+Print`
`↪GCDateStamps`. Użyteczna może być też opcja `-Xloggc:<nazwa_pliku>`, która umożliwia zapisanie danych do pliku w celu późniejszej analizy offline.

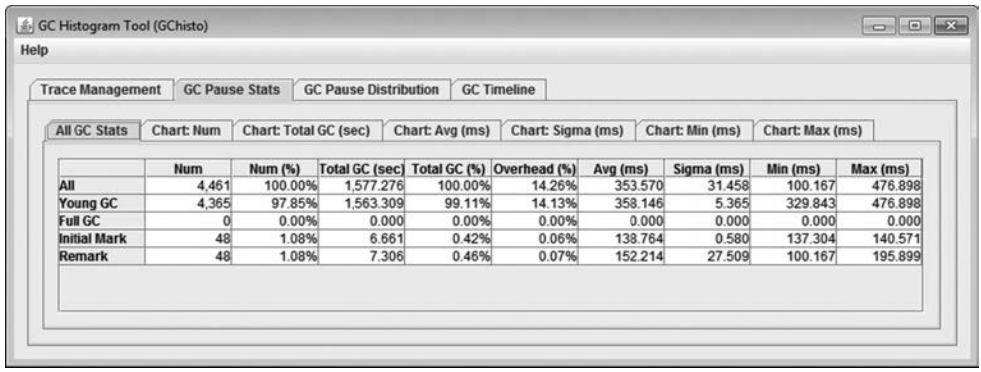
Analiza offline danych z procesów odzyskiwania pamięci

Celem przeprowadzania analizy offline jest podsumowanie danych dotyczących odzyskiwania danych oraz szukanie wzorców w układach danych. Analiza offline danych z procesów odzyskiwania pamięci może być przeprowadzana na szereg różnych sposobów, takich jak np. import danych do arkusza kalkulacyjnego lub skorzystanie z narzędzia do tworzenia wykresów. Jednym z narzędzi przeznaczonych do przeprowadzania analizy offline jest GCHisto. Jest to bezpłatne narzędzie, które możesz pobrać ze strony *http://gchisto.dev.java.net*. GCHisto odczytuje zapisane w pliku dane z odzyskiwania pamięci i prezentuje je zarówno w formie tabelarycznej, jak i w postaci graficznej. Na rysunku 4.1 przedstawiono tabelaryczne podsumowanie z zakładki *GC Pause Stats* (statystyki przestojów mechanizmów odzyskiwania pamięci).

Zakładka *GC Pause Stats* dostarcza informacji, takich jak liczba, obciążenie i czas trwania procesów odzyskiwania pamięci. Dodatkowe opcje w tej zakładce służą do zawężenia obszaru analizy do jednej z wyżej wymienionych kategorii.

Wszystkie procesy odzyskiwania pamięci oraz fazy tych procesów, które powodują przestoje typu „stop-the-world", zostały wyszczególnione w osobnym wierszu w tabeli, a w pierwszym wierszu podana została ich wartość całkowita. Na rysunku 4.1 przedstawiono dane dla równoczesnego mechanizmu odzyskiwania pamięci. Jak sobie pewnie przypominasz z rozdziału 3., równoczesny mechanizm odzyskiwania pamięci poza mniejszym (młodego pokolenia) i większym odzyskiwaniem pamięci generuje również dwa przestoje typu „stop-the-world". Są to znak inicjujący CMS oraz ponowne zaznaczanie CMS. Jeśli zaobserwujesz, że przestoje dla fazy znaku inicjującego oraz fazy ponownego zaznaczania są dłuższe niż przestoje dla procesu mniejszego odzyskiwania pamięci, jest to sugestia, że należy wykonać regulację JVM. Przestój wywołany przez każdą z tych dwóch faz powinien trwać krócej niż mniejsze odzyskiwanie pamięci.

Kup książkę Poleć książkę



	Num	Num (%)	Total GC (sec)	Total GC (%)	Overhead (%)	Avg (ms)	Sigma (ms)	Min (ms)	Max (ms)
All	4,461	100.00%	1,577.276	100.00%	14.26%	353.570	31.458	100.167	476.898
Young GC	4,365	97.85%	1,563.309	99.11%	14.13%	358.146	5.365	329.843	476.898
Full GC	0	0.00%	0.000	0.00%	0.00%	0.000	0.000	0.000	0.000
Initial Mark	48	1.08%	6.661	0.42%	0.06%	138.764	0.580	137.304	140.571
Remark	48	1.08%	7.306	0.46%	0.07%	152.214	27.509	100.167	195.899

Rysunek 4.1. Statystyki przestojów mechanizmów odzyskiwania pamięci w GCHisto

Ponieważ przepustowościowy mechanizm pamięci generuje przestoje „stop-the-world" tylko dla procesów odzyskiwania pamięci, w zakładce *GC Pause Stats* narzędzia GCHisto dla tego mechanizmu znajdziesz tylko dane dotyczące mniejszego i pełnego odzyskiwania pamięci.

Liczba procesów mniejszego odzyskiwania pamięci w porównaniu z liczbą procesów pełnego odzyskiwania pamięci daje wyobrażenie o częstotliwości uruchamiania procesów pełnego odzyskiwania pamięci. Ta informacja wraz z danymi na temat długości przestojów na pełne odzyskiwanie pamięci może być oceniana pod względem wymagań aplikacji dotyczących czasu trwania procesów pełnego odzyskiwania pamięci.

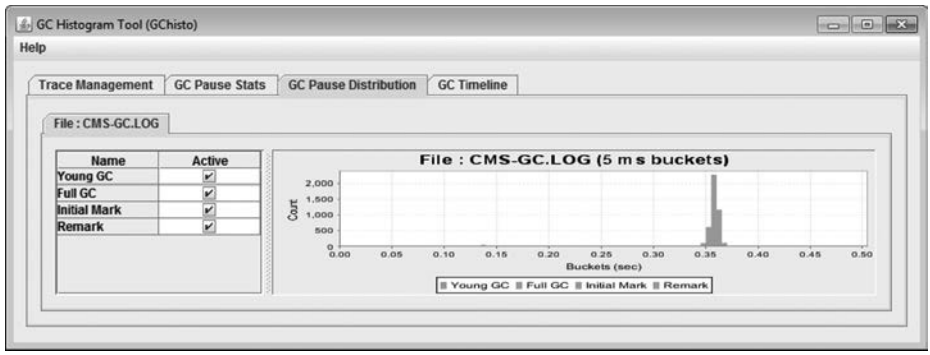
Obciążenie, jakie powoduje działanie mechanizmu odzyskiwania pamięci (kolumna `Overhead (%)`), jest wskaźnikiem tego, jak dobrze mechanizm został wyregulowany. Zasadniczo obciążenie dla równoczesnego mechanizmu odzyskiwania pamięci powinno wynosić mniej niż 10%. Teoretycznie możliwe jest osiągnięcie wartości z przedziału 1 – 3%. Z kolei dla przepustowościowego mechanizmu odzyskiwania pamięci obciążenie w okolicach 1% wskazuje na dobrze wyregulowany mechanizm. Wartość obciążenia 3% i więcej dla tego mechanizmu jest wskazaniem, że regulacja mechanizmu odzyskiwania pamięci może poprawić wydajność aplikacji. Ważne jest, aby zdawać sobie sprawę, że istnieje zależność pomiędzy obciążeniem mechanizmu odzyskiwania pamięci a rozmiarem sterty Java. Im większa sterta Java, tym większe możliwości do zmniejszenia obciążenia generowanego przez mechanizm odzyskiwania pamięci. Uzyskanie najmniejszego możliwego obciążenia dla określonego rozmiaru sterty Java wymaga regulacji JVM.

Na rysunku 4.1 obciążenie dla mechanizmu odzyskiwania pamięci wynosi nieco ponad 14%. Stosując się do ogólnych wskazówek opisanych wyżej, możemy przyjąć, że regulacja JVM prawdopodobnie zredukuje to obciążenie.

Maksymalne czasy przestojów umieszczone w ostatniej kolumnie po prawej stronie mogą być analizowane pod względem wymagań aplikacji dotyczących najgorszych zakładanych opóźnień wywoływanych przez procesy odzyskiwania pamięci. Jeśli którykolwiek z maksymalnych czasów przestojów przekracza wymagania aplikacji, należy rozważyć regulację JVM. To, w jak dużym stopniu wymagania aplikacji zostały przekroczone oraz ile z przedstawionych wartości przekracza te wymagania, determinuje, czy regulacja JVM jest koniecznością.

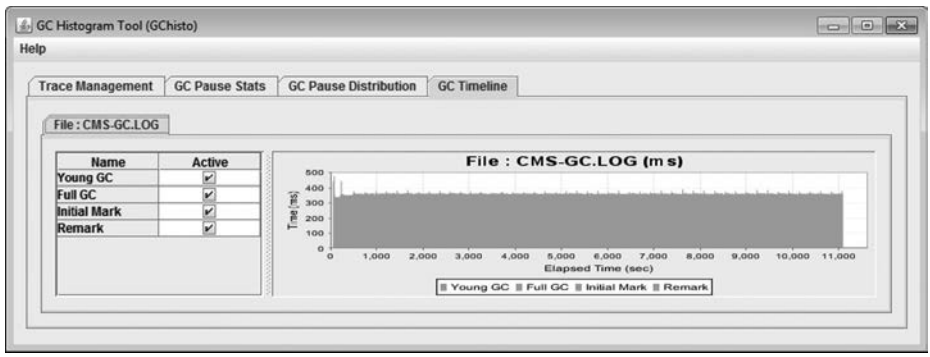
Wartości minimalne (kolumna `Min(ms)`), maksymalne (kolumna `Max(ms)`), średnie (kolumna `Avg(ms)`) oraz odchylenie standardowe (kolumna `Sigma(ms)`) dostarczają informacji na temat dystrybucji czasu przestojów. Dystrybucja czasu przestojów przedstawiona została w zakładce *GC Pause Distribution*, co pokazano na rysunku 4.2.

Kup książkę Poleć książkę



Rysunek 4.2. Dystrybucja przestojów dla mechanizmów odzyskiwania pamięci

Domyślny układ wykresu w zakładce *GC Pause Distribution* przedstawia rozkład przestojów dla wszystkich rodzajów procesów odzyskiwania pamięci. W oknie znajdującym się po lewej stronie wykresu możesz dodawać lub odejmować rodzaje procesów odzyskiwania pamięci, które mają być uwzględnione na wykresie. Oś y na wykresie reprezentuje liczbę przestojów, a oś x to czas trwania przestoju dla danego zdarzenia. Zasadniczo najwygodniej przeglądać osobno dane dotyczące procesów pełnego odzyskiwania pamięci, ponieważ są to zazwyczaj najdłuższe przestoje. Z kolei osobna analiza danych dla procesów mniejszego odzyskiwania pamięci daje możliwość zaobserwowania szerokiej różnorodności czasu przestojów. Szeroka dystrybucja czasu przestojów może wskazywać na duże wahania współczynnika alokacji oraz promocji obiektów. Jeśli zaobserwujesz szeroką dystrybucję czasu przestojów, powinieneś przejść do zakładki *GC Timeline* i odnaleźć najwyższe wartości dla aktywności procesów odzyskiwania pamięci. Przykład zamieszczono na rysunku 4.3.



Rysunek 4.3. Linia czasu w zakładce GC Timeline

Domyślny widok zakładki *GC Timeline* przedstawia przestoje dla wszystkich procesów odzyskiwania pamięci w funkcji czasu dla całego badanego okresu. Aby na dole wykresu na osi x widoczna była podziałka czasowa, musisz korzystać ze statystyk zawierających opcje `-XX:+PrintGCTimeStamps` i `-XX:+PrintGCDateStamps` lub skorzystać z opcji `-Xloggc`. Każdy z przestojów odzyskiwania pamięci został na wykresie zaznaczony „tickiem", aby zilustrować czas trwania przestoju (na osi y) oraz sytuację, kiedy przestój jest spowodowany uruchomieniem JVM (na osi *x*).

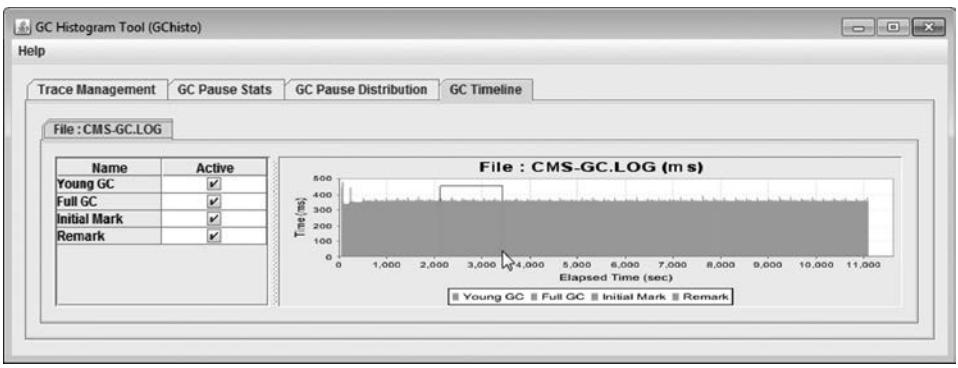
Istnieje kilka wzorców, których należy poszukiwać na wykresie z zakładki *GC Timeline*. Powinieneś np. zwrócić uwagę, kiedy mają miejsce procesy pełnego odzyskiwania pamięci i z jaką częstotliwością są przeprowadzane. Dla tej analizy warto ustawić jako rodzaj przestoju jedynie

Kup książkę Poleć książkę

pełne odzyskiwanie pamięci. Na wykresie czasu możesz zaobserwować, kiedy uruchamianie pełnego odzyskiwania pamięci jest związane z uruchamianiem JVM, co daje pogląd na to, w jakich momentach jest uruchamiane pełne odzyskiwane.

Ustawienie na wykresie jedynie przestojów dla procesów mniejszego odzyskiwania pamięci pozwala obserwować najwyższe i być może powtarzające się wartości dla czasu trwania odzyskiwania pamięci w danym przedziale czasu. Każde zaobserwowane szczyty wartości lub powtarzające się wzorce mogą zostać odwzorowane z powrotem w dzienniku zdarzeń aplikacji, aby zorientować się, co dzieje się w systemie w momencie, kiedy te zdarzenia mają miejsce. Przypadki użycia mechanizmu odzyskiwania pamięci w tych przedziałach czasu mogą stanowić sygnał do głębszego przyjrzenia się możliwościom redukcji alokacji oraz przechowywania obiektów. Zmniejszenie poziomu alokacji obiektów oraz ilości przechowywanych obiektów dla okresów największej aktywności mechanizmu odzyskiwania pamięci redukuje częstotliwość mniejszego odzyskiwania pamięci i potencjalnie może również zredukować częstotliwość pełnego odzyskiwania pamięci.

Interesujący Cię obszar wykresu dla linii czasu może zostać powiększony poprzez zaznaczenie za pomocą myszy wybranego fragmentu, co pokazano na rysunku 4.4.



Rysunek 4.4. Powiększanie obszaru wykresu w zakładce GC Timeline

Powiększenie umożliwia zawężenie obserwacji do obszaru określonego przedziału czasu, dzięki czemu możesz odczytać z wykresu czas każdego przestoju dla odzyskiwania pamięci. Aby przywrócić normalne proporcje wykresu, należy kliknąć prawym przyciskiem dowolne miejsce na obszarze wykresu i z menu kontekstowego wybrać opcję *Auto Range/Both Axes*.

Narzędzie GCHisto pozwala również na jednoczesne zaimportowanie kilku plików dziennika zdarzeń dla odzyskiwania pamięci. Służy do tego zakładka *Trace Management*. Przy załadowaniu wielu plików dziennika dla każdego pliku tworzona jest indywidualna zakładka, która umożliwia proste przełączanie pomiędzy danymi zawartymi w tych plikach. Jest to użyteczne, kiedy chcesz porównać dzienniki odzyskiwania zdarzeń dla różnych konfiguracji sterty Java lub dla różnych poziomów obciążenia aplikacji.

Narzędzia graficzne

Odzyskiwanie pamięci może być również monitorowane za pomocą narzędzi graficznych, które w porównaniu z analizą danych tekstowych mogą nieco ułatwić identyfikację trendów lub wzorców. Do monitorowania HotSpot VM można zastosować następujące narzędzia graficzne: JConsole, VisualGC oraz VisualVM. Narzędzie JConsole jest dystrybuowane z pakietem JDK dla Javy w wersji 5. oraz wersji kolejnych.

Kup książkę Poleć książkę

Narzędzie VisualGC zostało pierwotnie zaprojektowane i było dostarczane łącznie z jvmstat. Możesz je pobrać bezpłatnie ze strony *http://java.sun.com/performance/jvmstat*.

VisualVM jest projektem typu open source, który łączy w pojedynczym narzędziu kilka istniejących już wcześniej prostych funkcji służących do monitorowania i profilowania aplikacji Java. VisualVM jest dostępny w JDK dla Java 6 Update 6 i wersji późniejszych. Możesz go również pobrać bezpłatnie ze strony *http://visualvm.dev.java.net*.

JConsole

JConsole jest zgodnym z technologią JMX (ang. *Java Management Extensions*) narzędziem GUI, które podłącza się do działającej maszyny wirtualnej Javy dla wersji 5. Javy i nowszych. Aplikacje Java uruchomione na JVM dla Javy w wersji 5. muszą posiadać właściwość `-Dcom.sun.management.jmxremote`, aby umożliwić podłączenie JConsole. Tej właściwości nie wymagają aplikacje uruchomione na JVM dla Javy w wersji 6. i wersjach późniejszych. Poniższy przykład jest ilustracją, jak podłączyć JConsole do aplikacji demo o nazwie Java2Demo, która jest dostarczana z pakietem JDK. W przypadku Java 5 JDK aplikacja Java2Demo może być uruchomiona za pomocą następującej komendy wiersza poleceń.

W systemach Solaris i Linux:

```
$ <JDK_katalog_instalacji>/bin/java -Dcom.sun.management.jmxremote -jar <JDK_
katalog_instalacji>/demo/jfc/Java2D/Java2Demo.jar
```

`<JDK_katalog_instalacji>` to ścieżka do katalogu, w którym zainstalowany został pakiet Java 5 JDK.

W systemach Windows:

```
<JDK_katalog_instalacji>\bin\java -Dcom.sun.management.jmxremote -jar <JDK_
katalog_instalacji>\demo\jfc\Java2D\Java2Demo.jar
```

Do uruchomienia JConsole z JVM dla Javy w wersji 6. lub wersji późniejszych właściwość `-Dcom.sun.management.jmxremote` nie jest wymagana jako argument.

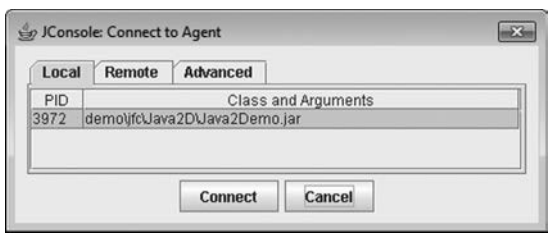
W systemach Solaris i Linux:

```
$ <JDK_katalog_instalacji>/bin/jconsole
```

W systemach Windows:

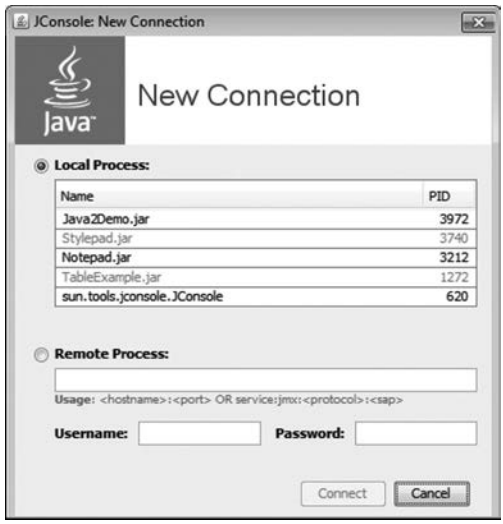
```
<JDK_katalog_instalacji>\bin\jconsole
```

Kiedy narzędzie JConsole zostanie uruchomione, automatycznie wykrywa i oferuje możliwości podłączenia do aplikacji Java działającej lokalnie lub zdalnie. Okno dialogowe połączenia różni się nieco w wersjach JConsole dostarczanych z Javą w wersji 5. i 6., co pokazano odpowiednio na rysunkach 4.5 i 4.6.



Rysunek 4.5. Okno dialogowe połączenia JConsole dla 5. wersji Javy

W JConsole dla Javy w wersji 5. monitorować można aplikacje znajdujące się na liście okna dialogowego połączenia, które zostały uruchomione za pomocą właściwości `-Dcom.sun.management.jmxremote`. Drugą kategorią aplikacji należących do tej grupy są aplikacje posiadające takie dane logowania użytkownika, co użytkownik, który uruchomił narzędzie JConsole.



Rysunek 4.6. Okno dialogowe połączenia JConsole dla 6. wersji Javy

Przy użyciu JConsole dla Javy w wersji 6. monitorować można aplikacje znajdujące się na liście okna dialogowego połączenia, które są aplikacjami Javy w wersji 6. oraz wersji 5. uruchomionymi za pomocą właściwości `-Dcom.sun.management.jmxremote`. Obie te grupy aplikacji muszą posiadać te same dane logowania użytkownika, co użytkownik, który uruchomił JConsole. Aplikacje Javy w wersji 5., które posiadają te same dane logowania użytkownika, a nie zostały uruchomione za pomocą właściwości `-Dcom.sun.management.jmxremote`, znajdują się na liście, ale są zaznaczone szarą czcionką i niedostępne.

Aby rozpocząć monitorowanie aplikacji w systemie lokalnym, należy wybrać parametry *Name* (nazwa aplikacji) oraz *PID* (identyfikator procesu aplikacji), a następnie kliknąć przycisk *Connect* (połącz). Monitoring zdalny (ang. *remote monitoring*) może być użyteczny, kiedy chcesz wyizolować z monitorowanego systemu wykorzystanie zasobów systemowych przez aplikację JConsole. Aplikacja monitorowana w zdalnym systemie musi zostać uruchomiona z włączoną funkcją zarządzania zdalnego (ang. *remote management*). Włączenie funkcji zarządzania zdalnego wiąże się z określeniem numeru portu służącego do komunikacji z monitorowaną aplikacją. Opcjonalnie można też ze względów bezpieczeństwa uruchomić protokół szyfrowania SSL. Informacje na temat sposobu uruchamiania funkcji zarządzania zdalnego znajdziesz w przewodnikach dotyczących zarządzania i monitorowania dla platform Java 5 SE i Java 6 SE:

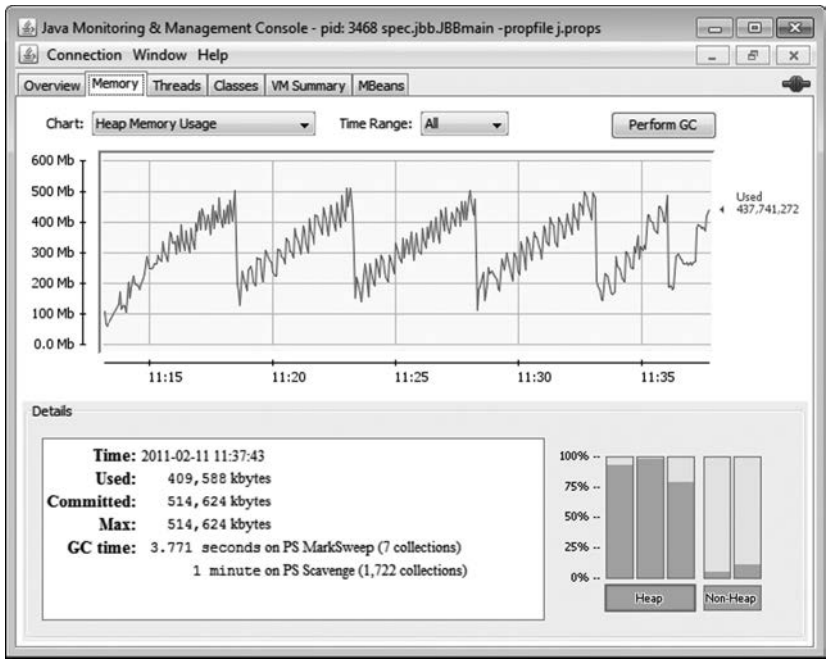
- Java SE 5 — *http://java.sun.com/j2se/1.5.0/docs/guide/management/index.html*;
- Java SE 6 — *http://java.sun.com/javase/6/docs/technotes/guides/management/toc.html*.

Wskazówka

Za pomocą narzędzia JConsole można równocześnie monitorować więcej niż jedną aplikację Java. W celu nawiązania połączenia z kolejną aplikacją wybierz menu *Connection/New Connection*, a następnie określ parę parametrów *Name* i *PID* dla tej aplikacji.

Kup książkę Poleć książkę

Po nawiązaniu przez narzędzie JConsole połączenia z aplikacją uzyskasz dostęp do sześciu zakładek przedstawiających różne statystyki. Domyślny widok okna JConsole różni się w wersjach dla Javy 5. i 6. W przypadku Javy w wersji 6. JConsole wyświetla graficzną reprezentację dla pamięci sterty, wątków, klas oraz wykorzystania procesora. Z kolei JConsole dla 5. wersji Javy podaje te same informacje, ale w formie tekstowej. Dla celów monitorowania procesów odzyskiwania pamięci w JVM najbardziej użyteczna jest zakładka *Memory* (pamięć). Zakładka ta wygląda tak samo zarówno w przypadku Javy w wersji 5., jak i 6. Na rysunku 4.7 przedstawiono zakładkę *Memory* narzędzia JConsole.



Rysunek 4.7. Zakładka Memory w JConsole

Zakładka Memory wykorzystuje wykresy w celu graficznego przedstawienia wykorzystania pamięci JVM w określonym przedziale czasu. Stosowane w JConsole nazwy przestrzeni, które składają się na stertę Java oraz nazwy pul pamięci, mogą różnić się w zależności od wersji JVM oraz stosowanego mechanizmu odzyskiwania pamięci. Nazwy te można jednak łatwo przyporządkować do następujących nazw przestrzeni HotSpot VM.

- **Eden** (ang. *eden space*). Pula pamięci, w której alokowane są prawie wszystkie obiekty Javy.
- **Przestrzeń ocalałych** (ang. *survivor space*). Pula pamięci zawierająca obiekty, które przetrwały przynajmniej jedno odzyskiwanie pamięci dla edenu.
- **Przestrzeń starego pokolenia lub przestrzeń zatrudnionych obiektów** (ang. *old space*, *tenured space*). Pula pamięci zawierająca obiekty, które przetrwały pewien próg wieku dla procesów odzyskiwania pamięci.
- **Przestrzeń stałego pokolenia** (ang. *permanent generation space*). Pula pamięci zawierająca wszystkie dane refleksyjne JVM, takie jak obiekty klas i metod. Jeśli monitorowana maszyna wirtualna Javy obsługuje udostępnianie danych klas, ta przestrzeń będzie podzielona na obszar tylko do odczytu oraz obszar z możliwością zapisu.

Kup książkę Poleć książkę

- **Pamięć podręczna kodu** (ang. *code cache*). Dotyczy HotSpot VM i zawiera pulę pamięci, która jest wykorzystywana przez kompilator JIT oraz służy do przechowywania skompilowanego kodu.

JConsole definiuje pamięć sterty jako kombinacje edenu, przestrzeni ocalałych oraz przestrzeni starego pokolenia. Pamięć nienależąca do sterty jest zdefiniowana jako kombinacja przestrzeni stałego pokolenia oraz pamięci podręcznej kodu. Możesz wyświetlać wykresy wykorzystania pamięci sterty (ang. *heap memory usage*) lub wykorzystania pamięci nienależącej do sterty (ang. *non-heap memory usage*), wybierając jedną z opcji rozwijanego menu *Chart* (wykres). Możesz również przeglądać wykresy dla wybranych przestrzeni. Ponadto klikając słupki opisane *Heap* (sterta) oraz *Non-Heap* (przestrzeń nienależąca do sterty), które znajdują się w prawym dolnym rogu, możesz przełączać się pomiędzy wykresami określonych przestrzeni *Heap* i *Non-Heap*. Nazwa przestrzeni lub puli pamięci, którą reprezentuje dany słupek, wyświetli się po najechaniu na słupek kursorem myszy.

Wzorzec, na który powinieneś zwrócić uwagę, to ten wzorzec, kiedy przestrzeń ocalałych pozostaje zapełniona przez dłuższy czas. Wskazuje on sytuację, w której przestrzenie ocalałych są przepełnione, a obiekty promowane do starego pokolenia, zanim będą miały możliwość osiągnięcia odpowiedniego wieku. Wyregulowanie rozmiaru przestrzeni młodego pokolenia może rozwiązać problemy z przepełnianiem się przestrzeni ocalałych.

Zakres czasowy wykresu wykorzystania pamięci można zmieniać, wybierając jedną z dostępnych opcji w rozwijanym menu *Time Range* (zakres czasowy).

W lewym dolnym rogu okna zakładki *Memory* wyświetlane są następujące bieżące metryki pamięci dla maszyny wirtualnej Javy.

- ***Used*** (pamięć wykorzystana). Ilość pamięci wykorzystywanej w danym momencie, włącznie z pamięcią zajętą przez obiekty Javy zarówno te osiągalne, jak i nieosiągalne.
- ***Committed*** (pamięć przydzielona). Gwarantowana ilość pamięci dostępnej dla JVM. Wartość ta może się zmieniać wraz z upływem czasu. Maszyna wirtualna Javy może uwalniać do systemu część przydzielonych zasobów pamięci, więc jej ilość może się zmniejszać w stosunku do wartości początkowej. Ilość pamięci przydzielonej jest zawsze większa lub równa ilości pamięci wykorzystanej.
- ***Max*** (pamięć maksymalna). Maksymalna ilość pamięci, która może być wykorzystana dla mechanizmu zarządzania pamięcią. Ilość tej pamięci może się zmieniać lub być niezdefiniowana. Alokacja pamięci może się nie powieść, jeśli JVM spróbuje wykorzystać więcej pamięci, niż określa to ilość pamięci przydzielonej, nawet jeśli ilość pamięci wykorzystanej jest mniejsza lub równa pamięci maksymalnej (np. kiedy w systemie zaczyna brakować pamięci wirtualnej).
- ***GC time*** (czas odzyskiwania pamięci). Łączny czas, który został przeznaczony na procesy odzyskiwania pamięci typu „stop-the-world", oraz całkowita liczba wywołań procesów odzyskiwania pamięci łącznie z bieżącym cyklami odzyskiwania pamięci. Każdy wiersz informacji w tym polu dotyczy osobnego mechanizmu odzyskiwania pamięci w JVM.

Narzędzie JConsole wyposażone zostało w jeszcze inne funkcje dotyczące monitorowania procesów odzyskiwania pamięci. Wiele z nich zostało opisanych w dokumentacji JConcole, którą możesz znaleźć na stronach:

- Java SE 5 — *http://java.sun.com/j2se/1.5.0/docs/guide/management/jconsole.html*;
- Java SE 6 — *http://java.sun.com/javase/6/docs/technotes/guides/management/jconsole.html*.

Kup książkę Poleć książkę

VisualVM

VisualVM jest graficznym narzędziem typu open source, które zostało opracowane w 2007 roku, wprowadzone w JDK dla Java 6 Update 7 i jest traktowane jako druga generacja narzędzia JConsole. VisualVM łączy kilka istniejących narzędzi programowych JDK oraz proste narzędzia monitorowania pamięci, takie jak JConsole. Ponadto w VisualVM znajdziesz funkcje profilowania znane z popularnego NetBeans Profiler. VisualVM zostało zaprojektowane do stosowania zarówno w środowiskach programistycznych, jak i środowiskach produkcyjnych. Poszerza ono zakres możliwości monitorowania oraz analizy wydajności dla platformy Java SE. VisualVM wykorzystuje również architekturę pluginów (wtyczek) NetBeans, co pozwala na proste dodawanie komponentów i wtyczek oraz daje możliwość zastosowania istniejących komponentów i wtyczek tego narzędzia do monitorowania i profilowania dowolnej aplikacji.

Do uruchomienia VisualVM wymagana jest platforma Java 6, ale samo narzędzie może być wykorzystane do monitorowania zdalnego lub lokalnego aplikacji Java 1.4.2, Java 5 lub Java 6. Istnieją jednak pewne ograniczenia dotyczące możliwości VisualVM w zależności od wersji Javy wykorzystywanej przez monitorowaną aplikację oraz tego, czy aplikacja działa lokalnie, czy zdalnie w stosunku do VisualVM. W tabeli 4.1 przedstawiono funkcje VisualVM dostępne dla określonej aplikacji Java działającej z konkretną wersją pakietu JDK.

Tabela 4.1. Zestawienie funkcji VisualVM

Funkcja	JDK 1.4.2 lokalnie i zdalnie	JDK 5.0 lokalnie i zdalnie	JDK 6.0 (zdalnie)	JDK 6.0 (lokalnie)
Przegląd	•	•	•	•
Właściwości systemu (przegląd)				•
Monitor	•	•	•	•
Wątki		•	•	•
Profilowanie				•
Zrzut wątku				•
Zrzut sterty				•
Włączenie zrzutu sterty na OOME				•
Przeglądarka MBean				•
Wrapper dla wtyczek JConsole (plugin)		•	•	•
VisualGC (plugin)	•	•	•	•

Jak już wspominaliśmy, VisualVM posiada również funkcje umożliwiające profilowanie. Chociaż profilowanie zostało opisane w rozdziale 5., funkcje zdalnego profilowania VisualVM zostały zawarte w tym rozdziale, ponieważ nie są skomplikowane i doskonale łączą się z czynnościami monitorowania.

Narzędzie VisualVM może być uruchomione w systemach operacyjnych Windows, Linux oraz Solaris za pomocą następującej komendy wiersza poleceń (zwróć uwagę, że nazwa polecenia to `jvisualvm`, a nie `visualvm`).

```
<JDK_katalog_instalacji>\bin\jvisualvm
```

(`<JDK_katalog_instalacji>` to ścieżka do katalogu, w którym zainstalowany został pakiet JDK 6 Update 6 lub nowszy).

Kup książkę Poleć książkę

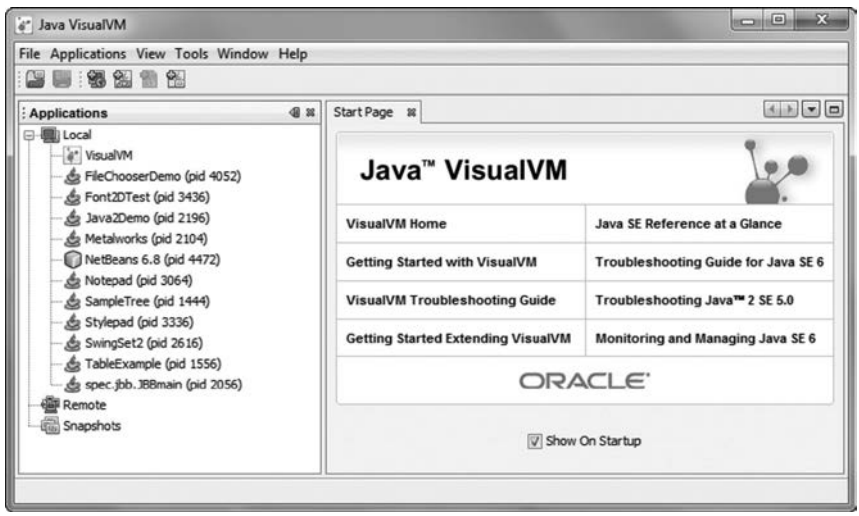
Jeśli posiadasz samodzielną wersję VisualVM pobraną ze strony *java.net*, jej uruchomienie w systemach Windows, Linux oraz Solaris odbywa się za pomocą następującej komendy wiersza poleceń (zwróć uwagę, że uruchomienie wersji samodzielnej VisualVM wymaga zastosowania polecenia `visualvm`, a nie `jvisualvm`, jak to ma miejsce dla VisualVM dostarczanego z pakietem JDK).

```
<VisualVM_katalog_instalacji>\bin\visualvm
```

(`<VisualVM_katalog_instalacji>` to ścieżka do katalogu, w którym zainstalowana została aplikacja VisualVM).

Alternatywnie możesz uruchomić VisualVM, korzystając z przeglądarki środowiska graficznego (np. Windows Explorer) poprzez dwukrotne kliknięcie pliku wykonywalnego aplikacji (*exe*), znajdującego się w katalogu instalacyjnym.

Początkowy widok okna VisualVM przedstawiono na rysunku 4.8. Okno to zostało podzielone na dwa obszary: *Applications* (aplikacje) oraz obszar wyświetlania monitorowanych statystyk.



Rysunek 4.8. VisualVM

Panel *Applications* ma postać drzewa, które posiada trzy podstawowe gałęzie. Gałąź *Local* zawiera listę lokalnych aplikacji Java, które mogą być monitorowane za pomocą VisualVM. Druga gałąź nazwana *Remote* przedstawia listę hostów zdalnych. Dla każdego zdalnego hosta wyświetlana jest lista aplikacji, które mogą być monitorowane za pomocą VisualVM. Gałąź *Snapshot* zawiera listę dostępnych plików zrzutów ekranu. VisualVM umożliwia robienie zrzutów ekranu dla określonego stanu aplikacji Java. Są one później zapisywane do plików i umieszczane na tej liście. Zrzuty ekranu mogą być użyteczne, kiedy chcesz zapisać ważne informacje dotyczące stanu aplikacji lub porównać ze sobą kilka zaobserwowanych sytuacji.

Lokalne aplikacje Java są automatycznie identyfikowane przez VisualVM w momencie ich uruchomienia lub w momencie uruchomienia VisualVM. Jak pokazano przykładowo na rysunku 4.8, VisualVM wykrył automatycznie aplikacje Java, które zostały pokazane w gałęzi *Local*. Kiedy uruchamiane są kolejne aplikacje Java, VisualVM również wykrywa je automatycznie i dodaje do listy w tej gałęzi. Gdy aplikacje są zamykane, zostają usunięte z listy.

Do monitorowania zdalnych aplikacji wymagana jest odpowiednia konfiguracja systemu, w którym te aplikacje działają. W zdalnym systemie musi być uruchomiona aplikacja jstatd daemon. Jest ona dystrybuowana w pakietach JDK Java 5 i Java 6, ale nie jest częścią pakietu JRE Java 5 czy

Kup książkę Poleć książkę

Java 6. Aplikację jstatd daemon znajdziesz w tym samym katalogu, w którym znajdują się jvisualvm oraz java launcher.

Jstatd daemon uruchamia aplikację Java RMI server, która monitoruje uruchamianie i zamykanie HotSpot VM oraz oferuje interfejs umożliwiający podłączenie i zdalne monitorowanie aplikacji Java takim narzędziom jak VisualVM. Jstatd daemon musi zostać uruchomiona z takimi samymi parametrami logowania użytkownika, co aplikacja Java, która ma być monitorowana. Ponieważ jstatd może ujawnić instrumentalizację JVM, musi posiadać menedżera zabezpieczeń i wymaga przy tym pliku polityki bezpieczeństwa. Powinieneś więc mieć na uwadze poziom dostępu zapewniany użytkownikom, aby nie narażać bezpieczeństwa monitorowanej maszyny wirtualnej Javy. Plik polityki bezpieczeństwa wykorzystywany przez jstatd musi być zgodny ze specyfikacją polityki bezpieczeństwa Javy. Poniżej podany został przykład pliku polityki bezpieczeństwa, który może być zastosowany z jstatd:

```
grant codebase "file:${java.home}/../lib/tools.jar" {
   permission java.security.AllPermission;
};
```

Wskazówka

Zwróć uwagę, że powyższy przykład pliku polityki bezpieczeństwa dopuszcza uruchamianie jstatd bez żadnych wyjątków bezpieczeństwa. Ta polityka jest mniej liberalna niż przyznanie wszelkich uprawnień każdej bazie kodów, ale jest bardziej liberalna niż polityka przyznająca minimalne uprawnienia do uruchomienia serwera jstatd. W celu dalszego ograniczenia dostępu można zdefiniować bardziej restrykcyjną politykę niż pokazano w tym przykładzie. Jeśli jednak nie ma możliwości spełnienia założeń bezpieczeństwa w pliku polityki bezpieczeństwa, najlepszym wyjściem jest stosowanie narzędzi monitorujących lokalnie zamiast monitorowania zdalnego i uruchamiania jstatd.

Zakładając, że polityka bezpieczeństwa jest zapisana w pliku *jstatd.policy*, możesz wykorzystać powyższy przykład polityki i uruchomić jstatd daemon za pomocą następującej komendy wiersza poleceń:

```
jstatd -J-Djava.security.policy=<ścieżka_do_pliku_polityki_bezpieczeństwa>/jstatd.policy
```

Wskazówka

Więcej szczegółów dotyczących konfiguracji jstatd znajdziesz na stronie *http://java.sun.com/javase/6/docs/technotes/tools/share/jstatd.html*.

Kiedy już jstatd daemon zostanie uruchomiony, na zdalnym systemie możesz zweryfikować, czy system lokalny może podłączyć się do jstatd daemon, stosując komendę `jps` z określeniem nazwy hosta zdalnego systemu. Komenda `jps` standardowo powoduje wyświetlenie aplikacji Java, które mogą być monitorowane. Użycie tej komendy z podaniem nazwy hosta powoduje próbę połączenia z jstatd daemon systemu zdalnego i wykrycie aplikacji Java, które mogą być monitorowane zdalnie. Jeśli nie podasz nazwy hosta dla komendy `jps`, otrzymasz listę aplikacji Java, które mogą być monitorowane lokalnie.

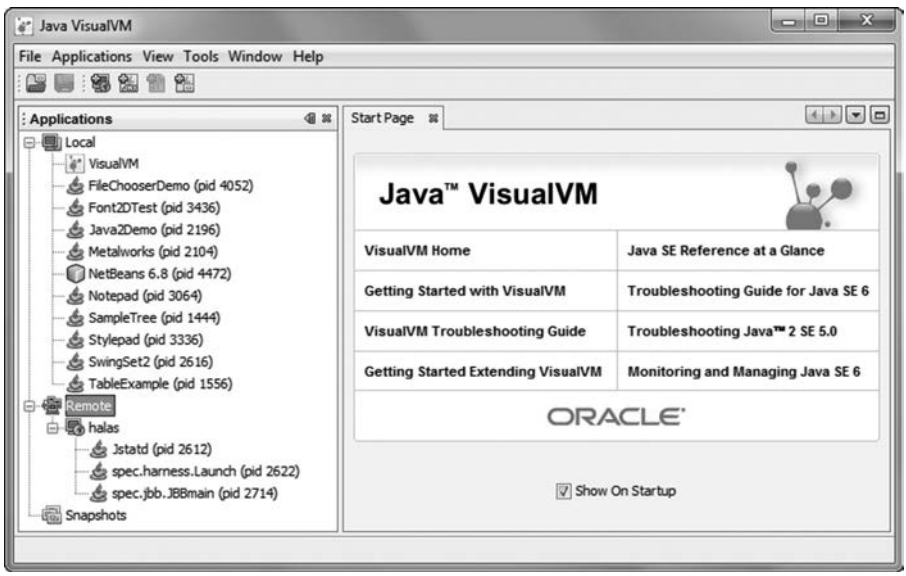
Załóżmy, że skonfigurowany przez Ciebie system zdalny, na którym uruchomiony został jstatd daemon, nazywa się „halas”. Aby zweryfikować w systemie lokalnym połączenie z systemem zdalnym, powinieneś wykonać komendę `jps` w następujący sposób:

Kup książkę Poleć książkę

```
$ jps halas
2622 Jstatd
```

Jeśli komenda `jps` zwraca wartość `Jstatd`, oznacza to, że jstatd daemon dla systemu zdalnego został skonfigurowany prawidłowo. Liczba znajdująca się przed wartością `Jstatd` w listingu określa id procesu dla jstatd daemon. Id procesu nie jest istotne dla weryfikacji zdalnego połączenia.

Aby monitorować zdalną aplikację za pomocą VisualVM, musisz skonfigurować dla tego narzędzia nazwę zdalnego hosta lub adres IP. W tym celu należy kliknąć prawym przyciskiem myszy gałąź *Remote* w panelu *Applications* i wpisać informacje dotyczące zdalnego hosta. Jeśli chcesz monitorować aplikacje Java na kilku hostach zdalnych, musisz dla każdego hosta skonfigurować jstatd daemon według procedury opisanej wcześniej. Następnie należy dodać w VisualVM informacje na temat każdego zdalnego hosta, który został skonfigurowany. VisualVM automatycznie wykrywa aplikacje Java, które mogą być monitorowane, i umieszcza je na liście. Pamiętaj, że każda zdalna aplikacja musi posiadać dane logowania użytkownika, takie same jak dane użytkownika, który uruchomił VisualVM oraz jstatd. Musi także posiadać uprawnienia określone w pliku polityki bezpieczeństwa jstatd. Na rysunku 4.9 przedstawiono VisualVM ze skonfigurowanym systemem zdalnym i listą aplikacji, które mogą być monitorowane.



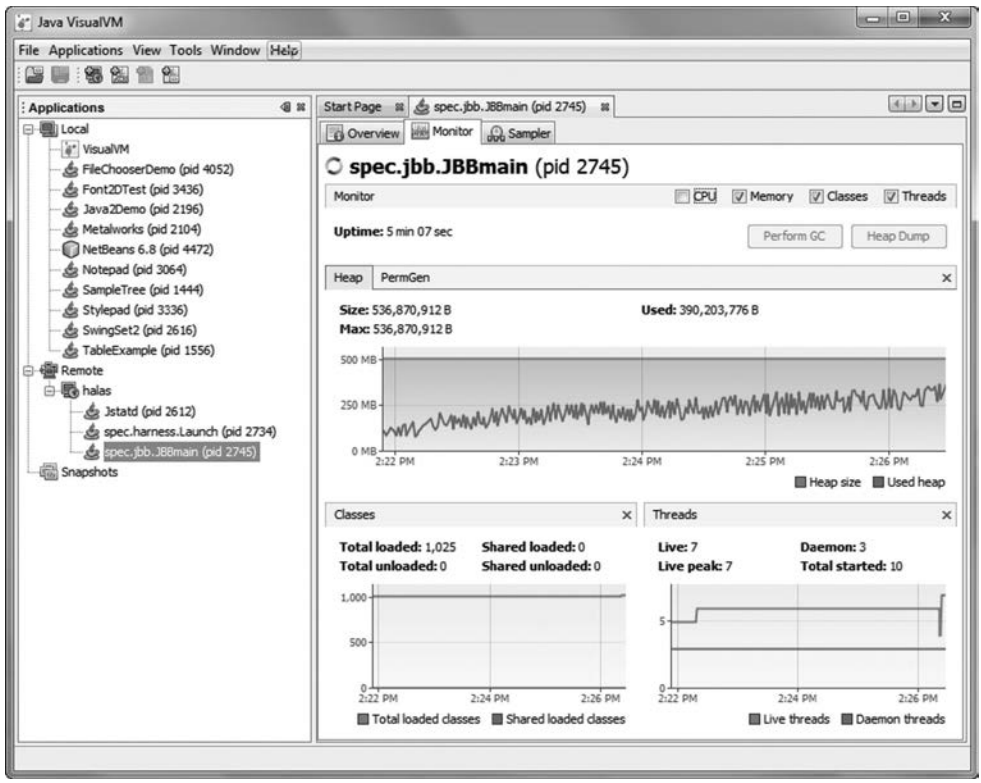
Rysunek 4.9. VisualVM skonfigurowany do monitorowania zdalnych aplikacji

Aby rozpocząć monitorowanie wybranej aplikacji, należy dwukrotnie kliknąć lewym przyciskiem myszy nazwę aplikacji lub reprezentującą ją ikonę w gałęziach *Local* lub *Remote*. Możesz również w tym samym miejscu kliknąć prawym przyciskiem myszy i wybrać z menu kontekstowego polecenie *Open* (otwórz). Każda z tych czynności spowoduje otwarcie w prawym panelu VisualVM okna zakładki dla wybranej aplikacji. W przypadku aplikacji lokalnych uruchomionych na platformie Java 6 (lub nowszej) widoczne będą również dodatkowe podzakładki.

Liczba okien podzakładek wyświetlanych w prawym panelu VisualVM zależy od wersji platformy Java aplikacji, od tego, czy jest to aplikacja zdalna, czy lokalna, oraz od tego, czy w VisualVM zainstalowane zostały jakiekolwiek dodatkowe wtyczki. W najgorszym razie dostępne są tylko dwie podzakładki: *Overview* (przegląd) oraz *Monitor* (monitor). Okno *Overview* zawiera wysokiego po-

Kup książkę Poleć książkę

ziomu przegląd monitorowanych aplikacji. Podawane są takie informacje jak: identyfikator procesu (ang. *process id*), nazwa hosta (ang. *host name*), na którym uruchomiona jest aplikacja, nazwa głównej klasy Java, argumenty przekazane do aplikacji, nazwa JVM, ścieżka do JVM, zastosowane flagi JVM, informacje, czy włączony jest zrzut sterty dla błędu *out of memory*, liczba wykonanych zrzutów wątku lub zrzutów sterty oraz właściwości systemu monitorowanej aplikacji, jeśli takie są dostępne. Okno *Monitor* wyświetla wykorzystanie sterty, wykorzystanie przestrzeni stałego pokolenia, informacje o załadowanych klasach oraz liczbę wątków. Przykład okna *Monitor* przedstawiającego monitorowaną aplikację działającą zdalnie na platformie Java 6 został pokazany na rysunku 4.10.



Rysunek 4.10. Okno Monitor w VisualVM

Jeśli dla zdalnej aplikacji zostało skonfigurowane połączenie JMX, możesz z poziomu okna *Monitor* uruchomić proces odzyskiwania pamięci lub wykonać zrzut sterty. Aby skonfigurować JMX dla zdalnej aplikacji, musi ona zostać uruchomiona co najmniej z poniższymi właściwościami systemu:

- `com.sun.management.jmxremote.port=<numer_portu>`;
- `com.sun.management.jmxremote.ssl=<true | false>`;
- `com.sun.management.jmxremote.authenticate=<true | false>`.

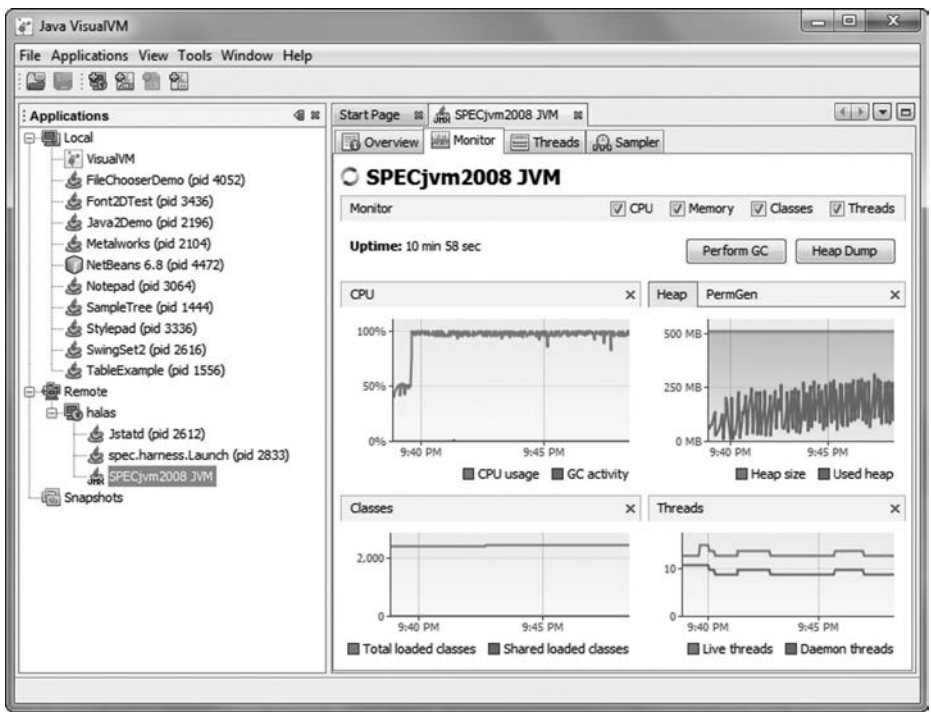
Żeby skonfigurować VisualVM tak, by łączył się ze zdalną aplikacją poprzez JMX, wybierz z menu *File* opcję *Add JMX Connection* (dodaj połączenie JMX). W oknie konfiguracji połączenia JMX trzeba w określonych polach wpisać poniższe informacje.

Kup książkę Poleć książkę

- W polu *Connection* należy wpisać `nazwa_hosta:<numer_portu>`. Jeśli np. zdalna aplikacja została uruchomiona na hoście *halas* i skonfigurowałeś właściwość `com.sun.management.jmxremote.port=4433`, to w polu *Connection* powinieneś wpisać `halas:4433`.
- Opcjonalnie możesz podać nazwę (ang. *display name*), która ma być wyświetlana w VisualVM w celu identyfikacji zdalnej aplikacji przez połączenie JMX. Domyślnie VisualVM używa jako nazwy wyświetlanej wartości z pola *Connection*.
- Jeśli włączyłeś uwierzytelnianie, konfigurując właściwość `com.sun.management.jmxremote.authenticate=true`, musisz w polach *Username* i *Password* podać odpowiednio nazwę użytkownika oraz hasło wymagane do połączenia zdalnego.

Więcej szczegółów na temat konfiguracji JMX dla zdalnego monitorowania aplikacji za pomocą VisualVM znajdziesz w dokumentacji VisualVM JMX na stronie *http://download.oracle.com/javase/6/docs/technotes/guides/visualvm/jmx_connections.html*.

Po skonfigurowaniu połączenia JMX dodatkowa ikona informująca o skonfigurowaniu zdalnego połączenia JMX do zdalnej aplikacji zostanie wyświetlona w panelu *Applications* narzędzia VisualVM. Skonfigurowanie połączenia JMX dla zdalnych aplikacji zwiększa możliwości monitorowania dla VisualVM. Przykładowo okno *Monitor* wyświetla dodatkowo wykorzystanie CPU przez aplikację oraz umożliwia inicjowanie procesu pełnego odzyskiwania pamięci i wykonywanie zrzutu sterty. Zostało to pokazane na rysunku 4.11.



Rysunek 4.11. Zdalny monitoring poprzez JMX

Poza zwiększeniem możliwości w oknie *Monitor* dostępna jest również dodatkowa podzakładka *Threads* (wątki). W oknie *Threads* pokazane są wątki aplikacji oznaczone różnymi kolorami, które określają ich status: działający, uśpiony, zablokowany, oczekujący lub rywalizujący o blokadę monitora. Standardowo okno *Threads* jest dostępne dla wszystkich aplikacji monitorowanych lokalnie.

Kup książkę Poleć książkę

Okno *Threads* oferuje informacje dotyczące najbardziej aktywnych wątków, które są w trakcie pozyskiwania i uwalniania blokad. Okno to może być użyteczne do obserwacji określonych zachowań wątków w aplikacji, szczególnie w sytuacji, kiedy statystyki z monitorowania systemu operacyjnego sugerują, że w aplikacji może dochodzić do rywalizacji o blokady.

Dodatkową opcją dostępną w oknie *Threads* jest możliwość wykonywania zrzutu wątku za pomocą przycisku *Thread Dump*. Kiedy pojawia się żądanie zrzutu wątku, VisualVM otwiera w kolejnej zakładce okno wyświetlające zrzut wątku oraz w panelu *Applications* dodaje pod wpisem dotyczącym monitorowanej aplikacji wpis na temat tego zrzutu. Zwróć uwagę, że jeśli zrzuty wątku nie zostaną zapisane, nie będą przechowywane i dostępne po zamknięciu VisualVM. Zrzut wątku można zapisać, klikając prawym przyciskiem myszy ikonę lub etykietę zrzutu wątku znajdującą się pod nazwą aplikacji na liście w panelu *Applications*. Zapisane zrzuty wątku mogą być potem w dowolnym momencie załadowane do VisualVM poprzez wybranie z menu *File* opcji *Load* i wskazanie katalogu, w którym zrzut wątku został zapisany.

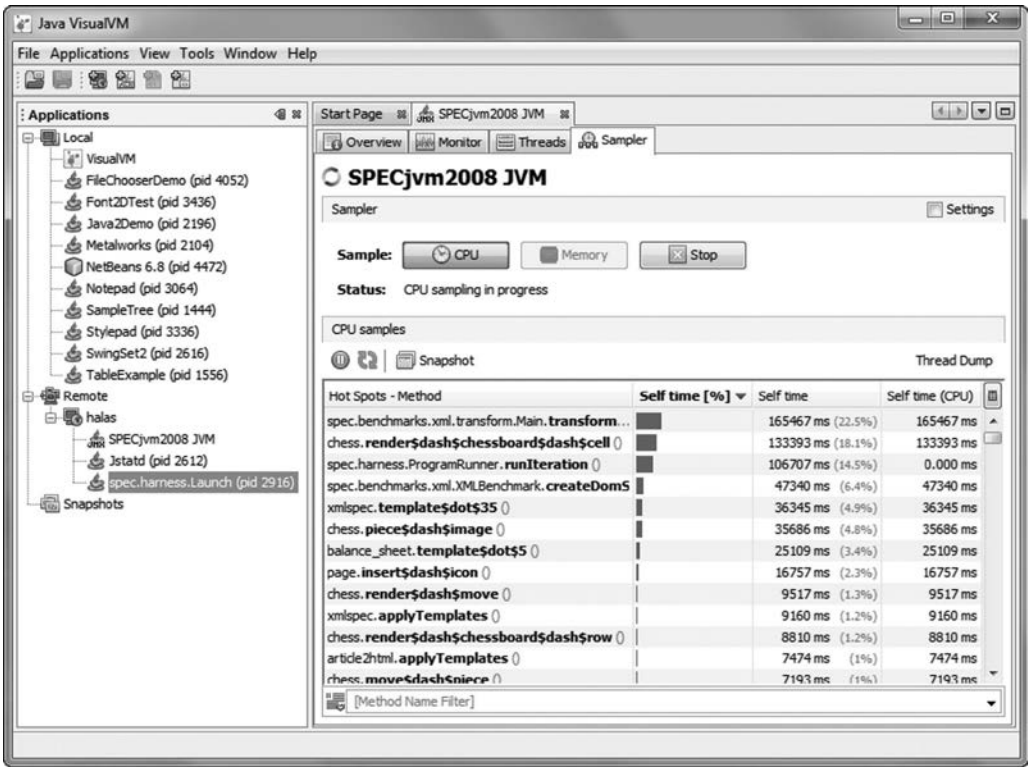
VisualVM oferuje również funkcje profilowania zarówno dla aplikacji zdalnych, jak i lokalnych. Funkcje lokalnego profilowania umożliwiają profilowanie CPU oraz pamięci dla aplikacji Java 6. Dla celów monitorowania użyteczne mogą być funkcje monitorowania wykorzystania CPU oraz wykorzystania pamięci w trakcie działania aplikacji. Musisz być jednak ostrożny, uruchamiając którąś z tych funkcji w systemie produkcyjnym, gdyż mogą one znacząco obciążyć działającą aplikację. Możliwość monitorowania wykorzystania CPU podczas pracy aplikacji pozwala uzyskać informacje na temat tego, które metody są najbardziej wykorzystywane podczas określonych zdarzeń. Przykładowo aplikacja GUI może doświadczać problemów z wydajnością tylko wtedy, kiedy otwarte jest określone okno tej aplikacji. Dlatego też możliwość monitorowania aplikacji GUI w momencie, w którym otwarte jest interesujące nas okno aplikacji, może być pomocna dla wyizolowania przyczyny problemu.

Profilowanie zdalne wymaga skonfigurowania połączenia JMX i jest ograniczone do profilowania CPU. Nie umożliwia ono natomiast profilowania pamięci. Możliwe jest za to wykonywanie zrzutów sterty z poziomu okna *Sampler* lub okna *Threads*. Zapisane uprzednio zrzuty sterty mogą zostać załadowane do VisualVM w celu przeprowadzenia analizy wykorzystania pamięci.

Aby rozpocząć zdalne profilowanie w VisualVM, wybierz i uruchom w panelu *Applications* zdalną aplikacje ze skonfigurowanym połączeniem JMX. Następnie przejdź do okna *Sampler* w prawym panelu i kliknij przycisk *CPU* w celu rozpoczęcia procesu profilowania. Na rysunku 4.12 przedstawiono wygląd okna *Sampler* po naciśnięciu przycisku *CPU*. W zestawieniu ilustrującym wykorzystanie CPU nazwa metody pochłaniającej najwięcej czasu umieszczona jest na szczycie listy. W drugiej kolumnie — *SelfTime %* (czas własny) — przedstawiony jest histogram czasu spędzonego w danej metodzie w stosunku do czasu poświęconego na inne metody. Następna kolumna — *SelfTime* — określa czas poświęcony na daną metodę wyrażony w milisekundach. Ostatnia kolumna — *Self Time (CPU)* — przedstawia czas CPU wykorzystany przez metodę. Każda z kolumn może być sortowana rosnąco lub malejąco poprzez kliknięcie nagłówka kolumny. Przy powtórnym kliknięciu nagłówka przełączasz się między sposobami sortowania (z rosnącego na malejący i odwrotnie).

Proces profilowania może być wstrzymany przyciskiem *Pause*, za pomocą którego również wznawia się wstrzymany proces. Do wykonywania zrzutów ekranu służy przycisk *Snapshot*. Po zrobieniu zrzut ekranu jest automatycznie wyświetlany przez VisualVM. Zrzut ekranu może zostać zapisany na dysk. Aby zapisać zrzut ekranu w celu podzielenia się informacjami z innym programistą, późniejszego załadowania pliku do programu lub porównania go z innymi zrzutami ekranu, należy wykonać eksport zrzutu do pliku, co pokazano na rysunku 4.13. Zapisany plik zrzutu ekranu może być załadowany do programów VisualVM lub NetBeans Profiler. W celu załadowania zrzutu ekranu do VisualVM należy z menu *File* wybrać opcję *Load*, a następnie uruchomić filtrowanie plików po typie *Profiler Snapshots (*.nps)*, aby odnaleźć zapisany profil.

Kup książkę Poleć książkę



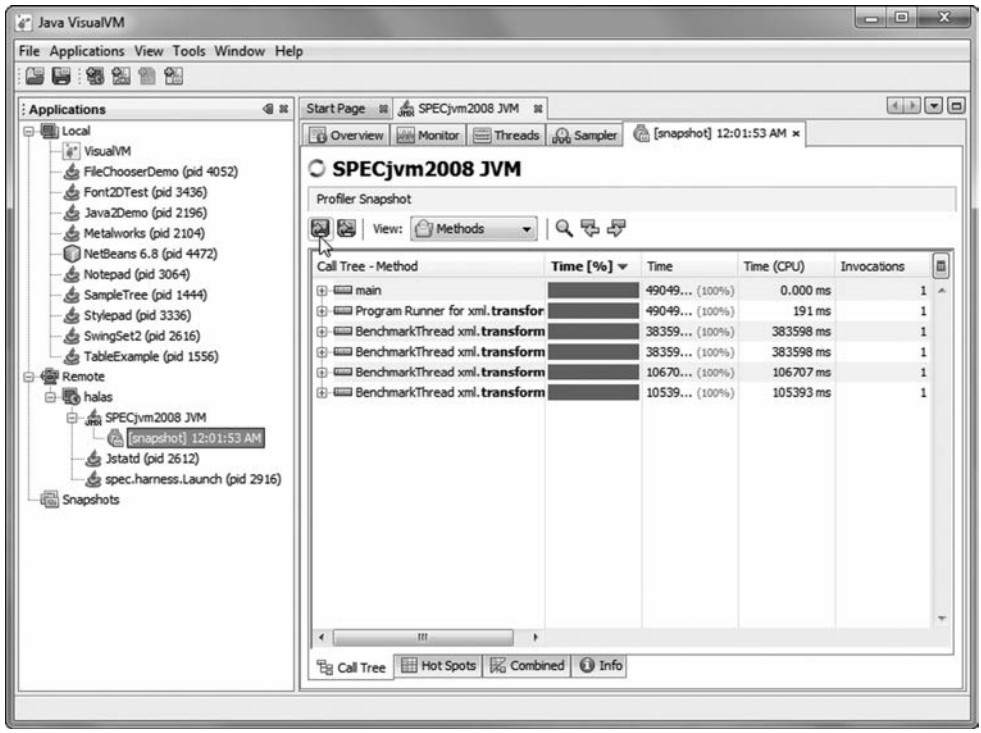
Rysunek 4.12. Zdalne profilowanie CPU

W oknie zrzutu ekranu wyświetlane jest drzewo wywołań, które przedstawia stosy wywołań (ang. *call stack*) dla wszystkich wątków przechwyconych w zrzucie ekranu. Każde drzewo wywołań można rozwinąć, by odczytać dane na temat stosu wywołań i metod, które pochłaniają najwięcej czasu oraz najbardziej wykorzystują procesor. Na dole okna znajduje się zakładka *Hot Spots*, która zawiera listę metod. Metoda pochłaniająca najwięcej czasu *Self Time* znajduje się na szczycie zestawienia. Możliwy jest również łączny podgląd zawartości zakładek *Call Tree* i *Hot Spots*. Jest on dostępny w zakładce *Combined*. Jeśli w zakładce *Combined* klikniesz wybrany stos wywołań w drzewie wywołań, tabela metod *Hot Spots* jest aktualizowana, tak aby pokazywać tylko metody dla wybranego stosu wywołań.

Dodatkowe informacje na temat profilowania aplikacji Java znajdziesz w rozdziale 5.

VisualVM umożliwia również ładowanie binarnych plików zrzutów sterty generowanych za pomocą jmap, JConsole lub w momencie wystąpienia błędu *OutOfMemoryError*, jeśli zastosowano opcję wiersza poleceń HotSpot VM `-XX:+HeapDumpOnOutOfMemoryError`. Binarny zrzut sterty to zrzut ekranu wszystkich obiektów w stercie JVM zrobiony w momencie generowania zrzutu sterty. Aby utworzyć binarny zrzut sterty za pomocą komendy `jmap` dla platformy Java 6, należy zastosować składnię `jmap -dump:format=b,file=<nazwa_pliku> <jvm pid>`, gdzie *<nazwa_pliku>* określa ścieżkę i nazwę pliku dla pliku binarnego zrzutu sterty, a *<jvm pid>* to identyfikator procesu dla JVM, na której uruchomiona została aplikacja. W przypadku platformy Java 5 komenda ta wygląda następująco: `jmap -heap:format=b <jvm pid>`, gdzie *<jvm pid>* to identyfikator procesu dla aplikacji Java. Komenda `jmap` dla Javy w wersji 5. zapisuje zrzut sterty do pliku o nazwie *heap.bin* w katalogu, w którym komenda `jmap` została wykonana. Narzędzie JConsole dla Javy w wersji 6.

Kup książkę Poleć książkę



Rysunek 4.13. Zapisywanie zrzutu ekranu

również umożliwia generowanie zrzutu stery za pomocą MBeana HotSpotDiagnostics. Utworzony plik binarny zrzutu sterty może być załadowany do VisualVM za pomocą opcji *Load* z menu *File* w celu przeprowadzenia analizy.

VisualGC

VisualGC jest wtyczką do narzędzia VisualVM, która umożliwia monitorowanie aktywności procesów odzyskiwania pamięci, ładowarki klas oraz kompilacji JIT. Pierwotnie VisualGC został zaprojektowany jako samodzielny program GUI. Obecnie może być stosowany samodzielnie lub jako wtyczka do VisualVM i pozwala monitorować maszyny wirtualne Javy dla platform Java 1.4.2, Java 5 oraz Java 6. Kiedy VisualGC został dostosowany jako wtyczka do VisualVM, dodano w nim kilka ulepszeń ułatwiających wykrywanie i podłączanie się do maszyn wirtualnych Javy. Przewagą wtyczki VisualGC nad wersją samodzielną GUI jest funkcja automatycznego wykrywania maszyn wirtualnych Javy, które mogą być monitorowane. Po wykryciu są one wyświetlane w VisualVM. W przypadku samodzielnej wersji GUI musisz określić identyfikator procesu aplikacji Java, którą chcesz monitorować i przekazać tę informację jako argument do *launchera* programu. Identyfikator procesu możesz sprawdzić za pomocą komendy `jps`. Przykład zastosowania komendy `jps` został opisany wcześniej w tym rozdziale jako element konfiguracji instalacyjnej jstatd daemon.

Wtyczka VisualGC jest dostępna w centrum wtyczek VisualVM, czyli w *Plug-in Center*. Centrum wtyczek otwiera się opcją *Plugins* w menu *Tools*. Wtyczka VisualGC powinna się znajdować na liście dostępnych wtyczek w zakładce *Available Plug-ins*.

Samodzielną wersję VisualGC GUI możesz pobrać ze strony *http://java.sun.com/performance/jvmstat/#Download*.

Kup książkę Poleć książkę

Bez względu na to, czy korzystasz z wersji samodzielnej VisualGC, czy z wtyczki, aplikacja monitorowana lokalnie musi posiadać te same dane logowania użytkownika, co VisualGC. W przypadku monitorowania aplikacji zdalnej te same dane użytkownika musi posiadać zarówno jstatd daemon, jak i monitorowana aplikacja. Sposób konfiguracji oraz uruchomienia jstatd daemon zostały opisane wcześniej w tym rozdziale.

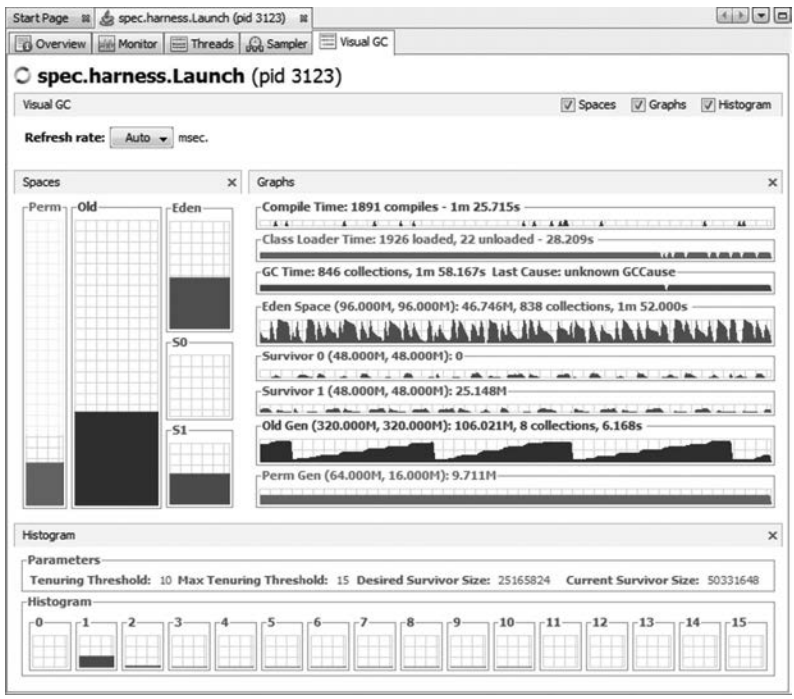
Stosowanie VisualGC jako wtyczki do programu VisualVM jest opisane w tym podpunkcie, ponieważ jest prostsze niż korzystanie z wersji samodzielnej VisualGC. Ponadto VisualVM oferuje również inne zintegrowane funkcje monitorowania.

Po dodaniu wtyczki VisualGC do aplikacji VisualVM podczas monitorowania wybranej aplikacji z listy w panelu *Applications*, w prawym panelu dostępne jest dodatkowe okno z etykietą VisualGC (patrz rysunek 4.14).



Rysunek 4.14. Dodatkowa zakładka z oknem VisualGC

Zakładka VisualGC wyświetla dwa lub trzy panele w zależności od stosowanego mechanizmu odzyskiwania pamięci. Dla przepustowościowego mechanizmu odzyskiwania pamięci wyświetlane są dwa panele, *Spaces* (przestrzenie) oraz *Graphs* (wykresy). Jeśli stosowany jest równoczesny lub szeregowy mechanizm odzyskiwania pamięci pod panelami *Spaces* i *Graphs* umieszczony jest trzeci panel o nazwie *Histogram* (histogram). Na rysunku 4.15 przedstawiono okno VisualGC zawierające wszystkie trzy panele.



Rysunek 4.15. VisualGC

Kup książkę Poleć książkę

Każdy z tych trzech paneli może być dodany lub usunięty z okna zakładki VisulaGC poprzez zaznaczenie lub usunięcie zaznaczenia odpowiadających im pozycji umieszczonych w prawym górnym rogu.

Panel *Spaces* zawiera graficzne przedstawienie przestrzeni odzyskiwania pamięci oraz wykorzystania tych przestrzeni. Panel jest podzielony na trzy pionowe sekcje, z których każda symbolizuje jedną z przestrzeni odzyskiwania pamięci: *Perm* (przestrzeń stałego pokolenia), *Old* lub *Tenured* (przestrzeń starego pokolenia) oraz przestrzeń młodego pokolenia składającą się z edenu oraz dwóch przestrzeni ocalałych (*S0* i *S1*). Wielkość rozmiarów reprezentujących wymienione przestrzenie jest proporcjonalna do maksymalnych pojemności tych przestrzeni alokowanych przez maszynę wirtualną Javy. Bieżący poziom wykorzystania edenu w stosunku do maksymalnej pojemności symbolizowany jest stopniem wypełnienia jego obszaru określonym kolorem. Dla każdej przestrzeni odzyskiwania pamięci zastosowano indywidualny kolor wypełnienia, który jest wykorzystywany konsekwentnie również w panelach *Graphs* i *Histogram*.

System zarządzania pamięcią w HotSpot VM może zwiększyć lub zmniejszyć obszar sterty po odzyskiwaniu pamięci, jeśli wartości ustawione dla `-Xmx` i `-Xms` są różne. Osiąga się to przez zastrzeżenie pamięci dla żądanego maksymalnego rozmiaru sterty Java, ale przy jednoczesnym przydzieleniu pamięci rzeczywistej jedynie do wielkości aktualnie wymaganej. Stosunek pomiędzy pamięcią przydzieloną a pamięcią zarezerwowaną jest reprezentowany przez kolor siatki tła w obszarze każdej przestrzeni. Pamięć nieprzydzielona oznaczona jest jasnoszarym kolorem siatki tła, natomiast pamięć przydzielona — kolorem ciemnoszarym. W wielu przypadkach wykorzystanie przestrzeni może być niemal takie same jak ilość pamięci przydzielonej, co utrudnia dokładne określenie na siatce tła miejsca podziału przestrzeni na obszar pamięci przydzielonej i nieprzydzielonej.

Stosunek rozmiarów edenu i przestrzeni ocalałych dla obszaru młodego pokolenia w panelu *Spaces* jest z reguły stały. Obydwie przestrzenie ocalałych mają zazwyczaj ten sam rozmiar, a przestrzeń ich pamięci jest w pełni przydzielona. Obszar pamięci dla edenu może być tylko częściowo przydzielony, szczególnie we wczesnych cyklach życia aplikacji.

Kiedy uruchomiony za pomocą opcji `-XX:+UseParallelGC` lub `-XX:+UseParallelOldGC` przepustowościomy mechanizm odzyskiwania pamięci stosowany jest wraz z funkcją adaptacyjnego ustalania rozmiaru (włączoną domyślnie), wzajemny stosunek rozmiarów przestrzeni młodego pokolenia może zmieniać się z upływem czasu. Włączona funkcja adaptacyjnego ustalania rozmiaru może doprowadzić do sytuacji, w której rozmiary przestrzeni ocalałych nie będą identyczne, a całkowity rozmiar przeznaczony na przestrzeń młodego pokolenia zostanie dynamicznie rozlokowany pomiędzy jego trzema przestrzeniami składowymi. W przypadku takiej konfiguracji obszary reprezentujące w panelu *Spaces* przestrzenie ocalałych oraz wypełnienie symbolizujące wykorzystanie tych obszarów mają rozmiar relatywnie odpowiadający bieżącemu, a nie maksymalnemu rozmiarowi całej przestrzeni. Kiedy maszyna wirtualna Javy adaptacyjnie zmienia rozmiar przestrzeni młodego pokolenia, obszar reprezentujący tę przestrzeń w panelu *Spaces* jest odpowiednio aktualizowany.

Panel *Spaces* należy obserwować pod kątem występowania kilku określonych sytuacji. Przykładowo powinieneś zwracać uwagę na prędkość zapełniania się przestrzeni edenu. Każde zapełnienie tej przestrzeni oraz następujący po nim spadek zapełnienia reprezentują mniejsze odzyskiwanie pamięci. Częstotliwość występowania tego zdarzenia jest częstotliwością uruchamiania procesu mniejszego odzyskiwania pamięci. Obserwując przestrzenie ocalałych, możesz zauważyć, że przy każdym mniejszym odzyskiwaniu pamięci jedna z przestrzeni ocalałych jest zapełniona, a druga pusta. Ta obserwacja umożliwia zrozumienie sposobu, w jaki mechanizm odzyskiwania pamięci kopiuje żywe obiekty z jednej przestrzeni ocalałych do drugiej przy każdym procesie mniejszego odzyskiwania pamięci. Ważniejsze jednak jest monitorowanie przepełniania się przestrzeni ocalałych. Taką

Kup książkę Poleć książkę

sytuację można zidentyfikować, przyglądając się zapełnieniu przestrzeni ocalałych podczas mniejszego odzyskiwania pamięci. Jeśli przestrzenie ocalałych są w pełni lub prawie w pełni zapełnione po każdym mniejszym odzyskiwaniu pamięci oraz wzrasta zapełnienie przestrzeni starego pokolenia, przestrzenie ocalałych mogą się przepełniać. Zasadniczo wskazuje to, że obiekty są promowane z młodego do starego pokolenia. Jeśli jednak są one promowane zbyt wcześnie lub zbyt szybko, może to ostatecznie doprowadzić do uruchomienia pełnego odzyskiwania pamięci. Przy pełnym odzyskiwaniu pamięci możesz zaobserwować spadek zapełnienia przestrzeni starego pokolenia. Częstotliwość występowania spadków wykorzystania przestrzeni starego pokolenia określa częstotliwość uruchamiania procesów pełnego odzyskiwania pamięci.

Panel *Graphs* pokazany na rysunku 4.15 znajduje się po prawej stronie okna VisualGC. Przedstawia on wykresy statystyk wydajności w funkcji czasu i oferuje historyczny przegląd danych. Panel ten wyświetla statystyki procesów odzyskiwania pamięci wraz ze statystykami kompilatora JIT i ładowarki klas. Te dwie ostatnie statystyki zostaną omówione w dalszej części rozdziału. Rozdzielczość osi poziomej dla każdego ze znajdujących się w tym panelu wykresów definiowana jest parametrem *Refresh Rate* znajdującym tuż nad panelem *Spaces*. Każda próbka dla historycznego wykresu danych w panelu *Graphs* zajmuje dwa piksele rozdzielczości ekranu. Wysokość każdego wykresu zależy od ilustrowanych danych statystycznych.

Panel *Graphs* zawiera następujące wykresy.

- ***Compile Time*** (czas kompilacji). Omówiony w dalszej części rozdziału.
- ***Class Loader Time*** (czas ładowarki klas). Omówiony w dalszej części rozdziału.
- ***GC Time*** (czas odzyskiwania pamięci). Wykres ilustrujący ilość czasu poświęcanego na działania związane z procesami odzyskiwania pamięci. Wysokość tego wykresu, czyli oś pionowa, nie jest skalowana do żadnej konkretnej wartości. Wartość różna od zera wskazuje na wykresie na działania mechanizmu odzyskiwania pamięci w ostatnim interwale czasowym. Wąski impuls wskazuje na relatywnie krótki czas trwania procesu, natomiast impuls szeroki to długi czas trwania. W opisie wykresu podano całkowitą liczbę procesów odzyskiwania pamięci oraz łączny czas trwania tych procesów od momentu uruchomienia aplikacji. Jeśli monitorowana maszyna wirtualna Javy gromadzi statystyki dotyczące przyczyn uruchamiania procesów odzyskiwania pamięci, w opisie podawana jest również przyczyna uruchomienia ostatniego procesu odzyskiwania pamięci.
- ***Eden Space*** (przestrzeń eden). Wykres przedstawiający wykorzystanie przestrzeni edenu w funkcji czasu. Wysokość tego wykresu jest stała, a domyślnie dane są skalowane zgodnie z aktualnym rozmiarem tej przestrzeni. Rozmiar przestrzeni edenu może się zwiększać lub zmniejszać w zależności od stosowanego w danym momencie mechanizmu odzyskiwania pamięci. W opisie wykresu znajduje się nazwa przestrzeni, podany w nawiasach maksymalny i bieżący rozmiar przestrzeni oraz bieżące zapełnienie przestrzeni. Dodatkowo podana jest tu łączna liczba procesów mniejszego odzyskiwania pamięci oraz ich skumulowany czas.
- ***Survivor 0 and Survivor 1*** (przestrzenie ocalałych *S0* i *S1*). Na tych wykresach przedstawiono wykorzystanie dwóch przestrzeni ocalałych w funkcji czasu. Wysokość każdego z tych wykresów jest stała, a domyślnie dane są skalowane zgodnie z bieżącym rozmiarem obu przestrzeni. Rozmiar przestrzeni ocalałych może zmieniać się wraz z upływem czasu w zależności od stosowanego mechanizmu odzyskiwania pamięci. W opisie każdego z wykresów podano nazwę przestrzeni, maksymalny i aktualny rozmiar przestrzeni w nawiasach oraz bieżące wykorzystanie przestrzeni.
- ***Old Gen*** (przestrzeń starego pokolenia). Wykres przedstawia wykorzystanie przestrzeni starego pokolenia w funkcji czasu. Wysokość wykresu jest stała, a domyślnie dane są

Kup książkę Poleć książkę

skalowane zgodnie z aktualnym rozmiarem tej przestrzeni. Rozmiar przestrzeni starego pokolenia może się zmieniać w zależności od stosowanego mechanizmu odzyskiwania pamięci. W opisie wykresu znajduje się nazwa przestrzeni, podany w nawiasach maksymalny i bieżący rozmiar przestrzeni oraz bieżące zapełnienie przestrzeni. Dodatkowo podana jest tu łączna liczba procesów pełnego odzyskiwania pamięci oraz ich skumulowany czas.

- ***Perm Gen*** (przestrzeń stałego pokolenia). Wykres przedstawia wykorzystanie przestrzeni stałego pokolenia w funkcji czasu. Wysokość wykresu jest stała, a domyślnie dane są skalowane zgodnie z aktualnym rozmiarem tej przestrzeni. Rozmiar przestrzeni stałego pokolenia może się zmieniać w zależności od stosowanego mechanizmu odzyskiwania pamięci. W opisie wykresu znajduje się nazwa przestrzeni, podany w nawiasach maksymalny i bieżący rozmiar przestrzeni oraz bieżące zapełnienie przestrzeni.

Panel *Histogram* pokazany na rysunku 4.15 znajduje się pod panelami *Spaces* i *Graphs* i jest wyświetlany tylko w podczas stosowania równoczesnego lub szeregowego mechanizmu odzyskiwania pamięci. W przypadku przepustowościowego odzyskiwania pamięci nie przechowuje się danych na temat wieku obiektów ocalałych, ponieważ stosowany jest tu inny mechanizm do zarządzania obiektami w przestrzeniach ocalałych. Z tego powodu panel *Histogram* nie jest wyświetlany, kiedy monitorowana maszyna wirtualna Javy stosuje przepustowościowy mechanizm odzyskiwania pamięci.

W panelu *Histogram* wyświetlane są statystyki dotyczące ocalałych obiektów oraz wieku obiektów. Jest on podzielony na dwa obszary: *Parameters* (parametry) i *Histogram*. W obszarze *Parameters* podane są informacje na temat bieżącego rozmiaru przestrzeni ocalałych oraz parametrów, które kontrolują promowanie obiektów z przestrzeni młodego pokolenia do przestrzeni starego pokolenia. Po każdym mniejszym odzyskiwaniu wzrasta wiek obiektu, pod warunkiem że obiekt pozostaje żywy. Jeśli wiek obiektu przekroczy określony wiek dla progu zatrudnienia (ang. *tenuring threshold age*), obiekt zostaje promowany do starego pokolenia. Wiek dla progu zatrudnienia jest wyliczany przez JVM podczas każdego procesu mniejszego odzyskiwania pamięci i wyświetlany w obszarze *Parameters* pod nazwą *Tenuring Threshold*. Maksymalny próg zatrudnienia (ang. *Max Tenuring Threshold*) również wyświetlany w tym obszarze to maksymalny wiek dla obiektów przechowywanych w przestrzeniach ocalałych. Obiekty są promowane z młodego do starego pokolenia na podstawie wartości parametru *Tenuring Threshold*, a nie *Max Tenuring Threshold*.

Często powtarzająca się sytuacja, w której próg zatrudnienia jest niższy niż maksymalny próg zatrudnienia, wskazuje, że obiekty są promowane z młodego do starego pokolenia zbyt szybko. Jest to zazwyczaj spowodowane przepełnieniem przestrzeni ocalałych. Jeśli przestrzeń ocalałych jest przepełniona, najstarsze obiekty są promowane do starego pokolenia do momentu, aż wykorzystanie przestrzeni ocalałych nie spadnie poniżej wartości *Desired Survivor Size* (pożądany rozmiar przestrzeni ocalałych) wyświetlanej w obszarze *Parameters*. Jak wcześniej wspominaliśmy, przepełnienie przestrzeni ocalałych może powodować zapełnienie przestrzeni starego pokolenia i w rezultacie wywołać uruchomienie procesu pełnego odzyskiwania pamięci.

Obszar *Histogram* w panelu *Histogram* wyświetla zrzut ekranu dla dystrybucji wieku obiektów w aktywnej przestrzeni ocalałych po ostatnim procesie mniejszego odzyskiwania pamięci. Jeśli mamy do czynienia z JVM dla Java 5 Update 6 lub nowszą, obszar ten zawiera szesnaście regionów o identycznym rozmiarze, z których każdy reprezentuje jeden z możliwych wieków obiektów. We wcześniejszych wersjach monitorowanej maszyny wirtualnej Javy będziemy mieli w obszarze *Histogram* trzydzieści dwa regiony. Każdy region reprezentuje 100% aktywnego obszaru przestrzeni ocalałych. Poziom wypełnienia regionu kolorem wskazuje procentowy stopień zapełnienia przestrzeni ocalałych obiektami o określonym wieku.

Kup książkę Poleć książkę

Podczas działania aplikacji możesz obserwować, jak długo żyjące obiekty przechodzą przez każdy z regionów wieku. Im większa przestrzeń zajmowana jest przez długo żyjące obiekty, tym większa będzie migracja obiektów pomiędzy regionami wieku. Kiedy próg zatrudnienia jest mniejszy niż maksymalny próg zatrudnienia, regiony reprezentujące wartości wyższe niż próg zatrudnienia pozostaną puste, ponieważ obiekty z tych regionów zostaną promowane do przestrzeni starego pokolenia.

Kompilator JIT

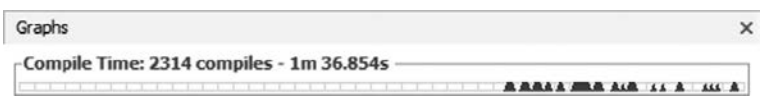
Istnieje kilka sposobów monitorowania aktywności kompilacji JIT dla maszyny wirtualnej HotSpot. Mimo że kompilacja JIT przyspiesza działanie aplikacji, wymaga określonych zasobów obliczeniowych, takich jak cykle CPU i pamięć. Z tego powodu użyteczne jest obserwowanie zachowania kompilatora JIT. Monitorowanie kompilacji JIT jest także użyteczne, kiedy chcesz zidentyfikować metody, które są optymalizowane, a w niektórych przypadkach deoptymalizowane lub optymalizowane ponownie. Metoda może być deoptymalizowana lub optymalizowana ponownie, jeśli kompilator JIT wykonał dla optymalizacji pewne wstępne założenia, które okazały się błędne. W takiej sytuacji kompilator JIT odrzuca taką optymalizację i ponownie wykonuje optymalizację metody na podstawie nowo pozyskanych informacji.

Do monitorowania kompilatora JIT HotSpot możesz wykorzystać opcję wiersza poleceń `-XX:+PrintCompilation`. Opcja ta generuje po jednej linii listingu dla każdej wykonanej kompilacji. Przykład listingu znajduje się poniżej:

```
 7       java.lang.String::indexOf (151 bytes)
 8% !    sun.awt.image.PNGImageDecoder::produceImage @ 960 (1920 bytes)
 9  !    sun.awt.image.PNGImageDecoder::produceImage (1920 bytes)
10       java.lang.AbstractStringBuilder::append (40 bytes)
11   n   java.lang.System::arraycopy (static)
12 s     java.util.Hashtable::get (69 bytes)
13   b   java.util.HashMap::indexFor (6 bytes)
14   made zombie  java.awt.geom.Path2D$Iterator::isDone (20 bytes)
```

Szczegółowe informacje dotyczące listingu dla opcji `-XX:+PrintCompilation` znajdziesz w dodatku A.

Istnieją też graficzne narzędzia do monitorowania aktywności kompilacji JIT, nie dostarczają one jednak tyle szczegółów, co opcja `-XX:+PrintCompilation`. Obecnie JConsole, VisualVM czy wtyczka VisualGC do VisualVM nie dostarczają informacji na temat tego, które metody są kompilowane przez kompilator JIT. Zapewniają one jedynie informacje o tym, kiedy kompilacja JIT ma miejsce. Najbardziej użytecznym narzędziem graficznym w tym przypadku może okazać się wykres *Compile Time* (czas kompilacji) z panelu *Graphs* okna VisualGC, pokazany na rysunku 4.16. Wykres ten przedstawia impulsy generowane aktywnością kompilacji JIT, które ułatwiają określenie miejsca występowania tych aktywności.



Rysunek 4.16. Wykres Compile Time w panelu Graphs wtyczki VisualGC

Wykres *Compile Time* w VisualGC pokazuje, jaka ilość czasu została przeznaczona na kompilację. Wysokość wykresu nie jest skalowana do żadnej konkretnej wartości. Impulsy na wykresie

Kup książkę Poleć książkę

reprezentują aktywności kompilacji JIT. Wąski impuls wskazuje na relatywnie krótki czas aktywności, a szeroki impuls — na długi czas aktywności. Obszary, na których nie ma żadnych impulsów, symbolizują brak aktywności kompilacji JIT. W opisie wykresu umieszczono całkowitą liczbę zadań kompilacji JIT oraz skumulowany czas, który upłynął na aktywnościach kompilacji.

Ładowanie klas

Wiele aplikacji korzysta ze zdefiniowanych przez użytkownika ładowarek klas zwanych *custom class loaders*. JVM ładuje klasy z ładowarki klas i może również zdecydować o wyładowaniu klasy. To, czy klasy są ładowane, czy rozładowywane, zależy od środowiska JVM Runtime oraz zastosowanych ładowarek klas. Monitorowanie aktywności ładowania klas może być użyteczne, szczególnie w aplikacjach, które korzystają z ładowarek klas zdefiniowanych przez użytkownika. Obecnie HotSpot VM ładuje wszystkie metadane klas do przestrzeni stałego pokolenia. Odzyskiwanie pamięci w przestrzeni stałego pokolenia przeprowadzane jest wtedy, kiedy przestrzeń ta się zapełni. Dlatego też monitorowanie zarówno aktywności ładowania klas, jak i wykorzystania przestrzeni stałego pokolenia może być istotne dla osiągnięcia przez aplikację wymagań dotyczących wydajności. Statystyki odzyskiwania pamięci wskazują, kiedy klasy są wyładowywane z przestrzeni stałego pokolenia.

Niewykorzystane klasy są wyładowywane z przestrzeni stałego pokolenia, kiedy wymagane jest dodatkowe miejsce na załadowanie innych klas. Do wyładowania klas z przestrzeni stałego pokolenia niezbędne jest pełne odzyskiwanie pamięci. W takim przypadku aplikacja może doświadczać problemów z wydajnością spowodowanych procesem pełnego odzyskiwania pamięci. Na poniższym listingu przedstawiono pełne odzyskiwanie pamięci, podczas którego przeprowadzone zostało wyładowywanie klas.

```
[Full GC[Unloading class sun.reflect.GeneratedConstructorAccessor3]
[Unloading class sun.reflect.GeneratedConstructorAccessor8]
[Unloading class sun.reflect.GeneratedConstructorAccessor11]
[Unloading class sun.reflect.GeneratedConstructorAccessor6]
 8566K->5871K(193856K), 0.0989123 secs]
```

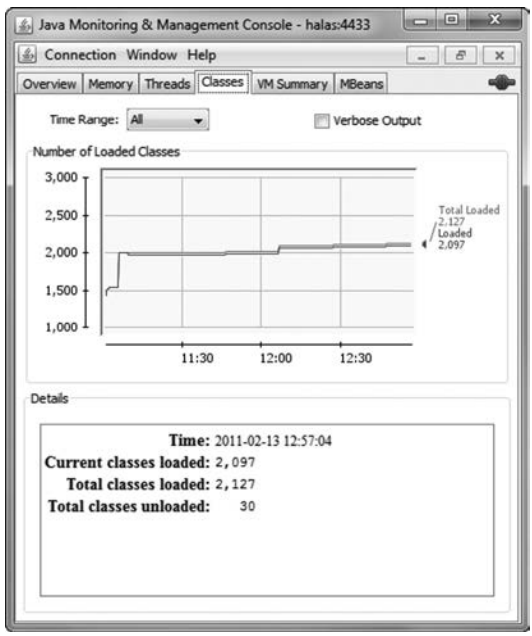
Powyższy listing wskazuje, że zostały wyładowane cztery klasy: `sun.reflect.Generated`
`↪ConstructorAccessor3`, `sun.reflect.Generated ConstructorAccessor8`, `sun.reflect.Generated`
`↪ConstructorAccessor11` oraz `sun.reflect.GeneratedConstructorAccessor6`. Raportowanie klas, które są wyładowywane podczas pełnego odzyskiwania pamięci, może dowodzić, że rozmiar przestrzeni stałego pokolenia powinien być większy lub większy powinien być początkowy rozmiar tej przestrzeni. Jeśli zauważysz, że podczas pełnego odzyskiwania pamięci zostały wyładowane klasy, powinieneś zastosować opcje wiersza poleceń `-XX:PermSize` oraz `-XX:MaxPermSize`, żeby zdefiniować rozmiar przestrzeni stałego pokolenia. Aby uniknąć pełnego odzyskiwania pamięci, które może poszerzyć lub zmniejszyć przydzielony rozmiar przestrzeni stałego pokolenia, ustaw takie same wartości dla `-XX:PermSize` oraz `-XX:MaxPermSize`. Zwróć uwagę, że jeśli włączone jest równoczesne odzyskiwanie pamięci dla stałego pokolenia, możesz zaobserwować wyładowywanie klas podczas cyklu równoczesnego odzyskiwania pamięci dla stałego pokolenia. Ponieważ cykl równoczesnego odzyskiwania pamięci dla stałego pokolenia nie jest wykonywany metodą „stop-the-world", aplikacja nie odczuwa skutków przestoju wywoływanego procesem odzyskiwania pamięci. Równoczesne odzyskiwanie pamięci dla starego pokolenia może być stosowane jedynie przez przeważnie-równoczesny mechanizm odzyskiwania pamięci, zwany CMS.

Kup książkę Poleć książkę

Wskazówka

Dodatkowe wskazówki i porady na temat regulacji rozmiaru przestrzeni stałego pokolenia wraz z instrukcją, jak uruchomić równoczesne odzyskiwanie pamięci dla stałego pokolenia, znajdziesz w rozdziale 7.

Monitorowanie ładowania klas umożliwiają takie narzędzia graficzne jak JConsole, VisualVM oraz wtyczka VisualGC dla VisualVM. Jednak aktualnie nie podają one nazw klas, które są ładowane lub wyładowywane. Jak pokazano na rysunku 4.17, zakładka *Classes* (klasy) w JConsole podaje liczbę aktualnie załadowanych klas, liczbę klas wyładowanych oraz całkowitą liczbę załadowanych klas.

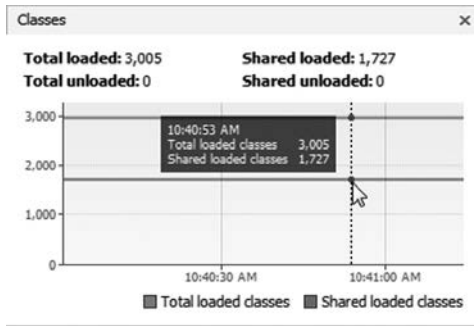


Rysunek 4.17. Całkowita liczba załadowanych klas i liczba aktualnie załadowanych klas

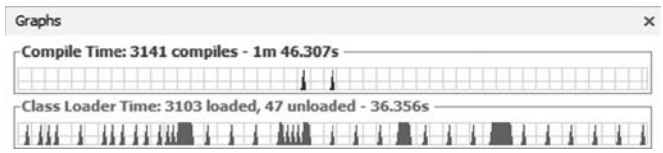
VisualVM również umożliwia monitorowanie aktywności ładowania klas w oknie *Classes* (klasy) zakładki *Monitor*. Okno to pokazuje całkowitą liczbę załadowanych klas oraz liczbę załadowanych klas współdzielonych. Możesz tutaj sprawdzić, czy w monitorowanie w maszynie wirtualnej Javy zostało włączone udostępnianie danych klas (ang. *class data sharing*). Funkcja udostępniania danych klas polega na współdzieleniu klas pomiędzy kilkoma maszynami wirtualnymi Javy działającymi na tym samym systemie. Pozwala to zmniejszyć wykorzystanie pamięci przez JVM-y. Jeśli monitorowana maszyna wirtualna Javy korzysta z udostępnianych klas, na wykresie poza poziomą linią symbolizującą całkowitą liczbę załadowanych klas pojawi się pozioma linia reprezentująca liczbę klas współdzielonych. Tego typu sytuację przedstawiono na rysunku 4.18.

Monitorowanie aktywności ładowania klas jest także możliwe na wykresie *Class Loader Time* w panelu *Graphs* wtyczki VisualGC, co pokazano na rysunku 4.19.

Kup książkę Poleć książkę



Rysunek 4.18. Zaobserwowane współdzielenie klas



Rysunek 4.19. Aktywność ładowania klas w VisualGC

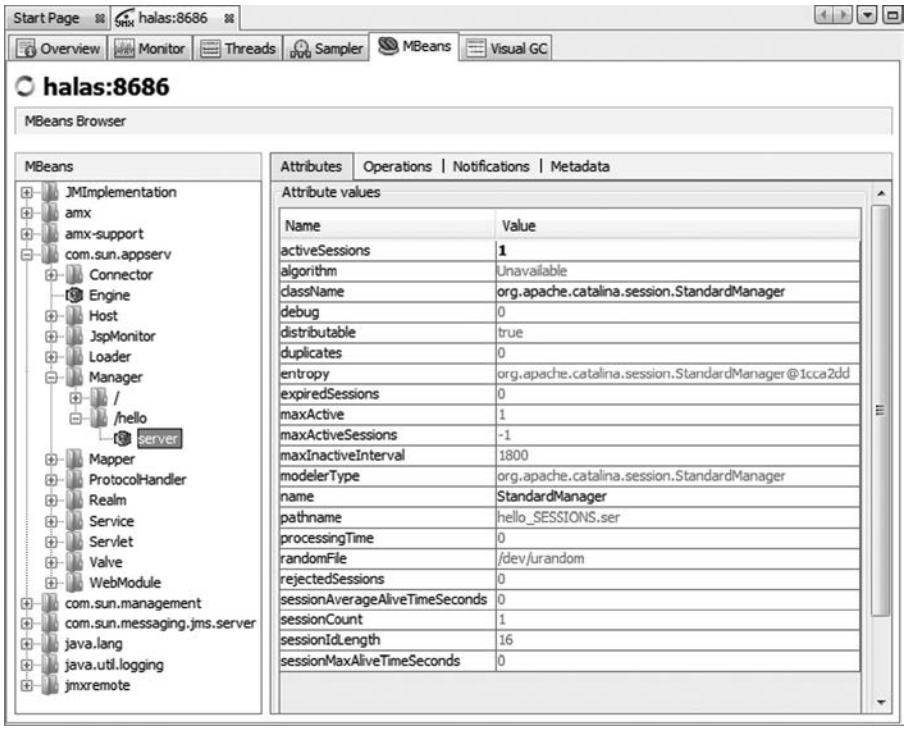
Impulsy na wykresie *Class Loader Time* symbolizują aktywność ładowania lub wyładowywania klas. Wąski impuls wskazuje na krótki czas aktywności, a szeroki impuls — na długi czas aktywności. Brak impulsu oznacza brak aktywności ładowania klas. W opisie wykresu podana została liczba załadowanych klas, liczba wyładowanych klas oraz łączny czas dla ładowania klas od momentu uruchomienia aplikacji. Obserwowanie na wykresie *Class Loader Time* impulsów, które bezpośrednio odpowiadają impulsom na znajdującym się poniżej w tym panelu wykresie *GC Time*, wskazuje na aktywność mechanizmu odzyskiwania pamięci w tym samym czasie. Może to być spowodowane przeprowadzaniem procesu odzyskiwania pamięci dla stałego pokolenia JVM.

Monitorowanie aplikacji Java

Monitorowanie na poziomie aplikacji z reguły wiąże się z obserwowaniem dziennika zdarzeń (logów), który zawiera interesujące nas zdarzenia lub instrumentalizację zapewniającą określony poziom informacji na temat wydajności aplikacji. Niektóre aplikacje mogą również posiadać wewnętrzne funkcje monitorowania i zarządzania wbudowane za pomocą MBeanów poprzez interfejsy monitorowania i zarządzania API dla Java SE. Te MBeany mogą być przeglądane i monitorowane za pomocą narzędzi zgodnych z technologią JMX, takich jak JConsole, lub za pomocą wtyczki VisualVM-MBeans dla VisualVM. Wtyczkę tę można znaleźć w centrum wtyczek w opcji *Plugins* dla menu *Tools*.

GlassFish Server Open Source Edition (zwany dalej GlassFish) posiada dużą liczbę atrybutów, które mogą być monitorowane za pomocą MBeanów. Korzystając z JConsole lub VisualVM do monitorowania instancji serwera aplikacji, GlassFish umożliwia przeglądanie MBeanów wraz z ich atrybutami i operacjami. Na rysunku 4.20 przedstawiono część z wielu MBeanów GlassFish w oknie *MBeans* narzędzia VisualVM korzystającego z wtyczki VisualVM-MBeans.

Kup książkę Poleć książkę



Rysunek 4.20. MBeany GlassFish

Po lewej stronie widać rozwiniętą listę MBeanów GlassFish w folderze *com.sun.appserv*.

Możliwości VisualVM mogą również zostać poszerzone o monitorowanie aplikacji Java, ponieważ narzędzie to zostało zbudowane na bazie architektury wtyczek NetBeans Platform. Wtyczki dla VisualVM mogą być tworzone, tak jakby były to wtyczki NetBeans. Przykładowo niestandardowa wtyczka do VisualVM umożliwiająca monitorowanie aplikacji Java może wykorzystać wiele bogatych funkcji NetBeans, włącznie z Visual Graph Library. Dla aplikacji, które mają udostępniać informacje dotyczące monitorowania wydajności, można przygotować odpowiednią wtyczkę VisualVM. Kilka istniejących już wtyczek jest dostępnych w centrum wtyczek VisualVM.

Aplikacje posiadające wtyczki JConsole mogą skorzystać z wtyczki VisualVM-JConsole, aby automatycznie zintegrować własne niestandardowe wtyczki JConsole z VisualVM.

Szybkie monitorowanie rywalizacji o blokady

Sztuczką często stosowaną przez autorów w celu uzyskania szybkiego poglądu na to, gdzie w aplikacji Java pojawia się rywalizacja o blokady, jest przechwycenie kilku zrzutów wątku za pomocą komendy JDK `jstack`. Takie podejście dobrze się sprawdza, kiedy wykonujesz wiele zadań monitorujących i potrzebujesz szybko przechwycić określone dane, bez konieczności poświęcania czasu na instalowanie i konfigurację aplikacji profilującej, która przeznaczona jest do bardziej szczegółowej analizy.

Poniższy listing z polecenia `jstack` pochodzi z aplikacji, która posiada zestaw wątków odczytu i zapisu dzielących jedną kolejkę. Wątki zapisu dodają zadania do kolejki, a wątki odczytu ujmują zadania z kolejki.

Kup książkę Poleć książkę

Zamieszczone zostały tylko istotne ślady stosu, aby zilustrować użyteczność komendy `jstack` do szybkiego odnajdywania blokad, o które toczy się rywalizacja. W listingu wątek `Read Thread-33` uzyskał z powodzeniem blokadę współdzielonej kolejki, która jest określona jako obiekt `Queue` o adresie `0x22e88b10`. Jest to zaznaczony wytłuszczonym drukiem fragment listingu **`locked <0x22e88b10> (a Queue)`**.

Wszystkie pozostałe przedstawione ślady stosu wątku czekają na uzyskanie tej samej blokady, którą posiada wątek `Read Thread-33`. Są to oznaczone wytłuszczonym drukiem fragmenty listingu **`waiting to lock <0x22e88b10> (a Queue)`**.

```
"Read Thread-33" prio=6 tid=0x02b1d400 nid=0x5c0 runnable
[0x0424f000..0x0424fd94]
   java.lang.Thread.State: RUNNABLE
      at Queue.dequeue(Queue.java:69)
      - locked <0x22e88b10> (a Queue)
      at ReadThread.getWorkItemFromQueue(ReadThread.java:32)
      at ReadThread.run(ReadThread.java:23)

"Writer Thread-29" prio=6 tid=0x02b13c00 nid=0x3cc waiting for monitor
entry [0x03f7f000..0x03f7fd94]
   java.lang.Thread.State: BLOCKED (on object monitor)
      at Queue.enqueue(Queue.java:31)
      - waiting to lock <0x22e88b10> (a Queue)
      at WriteThread.putWorkItemOnQueue(WriteThread.java:54)
      at WriteThread.run(WriteThread.java:47)

"Writer Thread-26" prio=6 tid=0x02b0d400 nid=0x194 waiting for monitor
entry [0x03d9f000..0x03d9fc94]
   java.lang.Thread.State: BLOCKED (on object monitor)
      at Queue.enqueue(Queue.java:31)
      - waiting to lock <0x22e88b10> (a Queue)
      at WriteThread.putWorkItemOnQueue(WriteThread.java:54)
      at WriteThread.run(WriteThread.java:47)

"Read Thread-23" prio=6 tid=0x02b08000 nid=0xbf0 waiting for monitor entry
[0x03c0f000..0x03c0fb14]
   java.lang.Thread.State: BLOCKED (on object monitor)
      at Queue.dequeue(Queue.java:55)
      - waiting to lock <0x22e88b10> (a Queue)
      at ReadThread.getWorkItemFromQueue(ReadThread.java:32)
      at ReadThread.run(ReadThread.java:23)

"Writer Thread-24" prio=6 tid=0x02b09000 nid=0xef8 waiting for monitor
entry [0x03c5f000..0x03c5fa94]
   java.lang.Thread.State: BLOCKED (on object monitor)
      at Queue.enqueue(Queue.java:31)
      - waiting to lock <0x22e88b10> (a Queue)
      at WriteThread.putWorkItemOnQueue(WriteThread.java:54)
      at WriteThread.run(WriteThread.java:47)

"Writer Thread-20" prio=6 tid=0x02b00400 nid=0x19c waiting for monitor
entry [0x039df000..0x039dfa14]
   java.lang.Thread.State: BLOCKED (on object monitor)
      at Queue.enqueue(Queue.java:31)
      - waiting to lock <0x22e88b10> (a Queue)
      at WriteThread.putWorkItemOnQueue(WriteThread.java:54)
      at WriteThread.run(WriteThread.java:47)
```

Kup książkę Poleć książkę

```
"Read Thread-13" prio=6 tid=0x02af2400 nid=0x9ac waiting for monitor entry
[0x035cf000..0x035cfd14]
   java.lang.Thread.State: BLOCKED (on object monitor)
      at Queue.dequeue(Queue.java:55)
      - waiting to lock <0x22e88b10> (a Queue)
      at ReadThread.getWorkItemFromQueue(ReadThread.java:32)
      at ReadThread.run(ReadThread.java:23)

"Read Thread-96" prio=6 tid=0x047c4400 nid=0xaa4 waiting for monitor
entry [0x06baf000..0x06bafa94]
   java.lang.Thread.State: BLOCKED (on object monitor)
      at Queue.dequeue(Queue.java:55)
      - waiting to lock <0x22e88b10> (a Queue)
      at ReadThread.getWorkItemFromQueue(ReadThread.java:32)
      at ReadThread.run(ReadThread.java:23)
```

Należy zwrócić uwagę, że wszystkie adresy blokady podane w nawiasach trójkątnych w zapisie szesnastkowym są takie same. W ten sposób blokady są unikatowo identyfikowane w listingu dla `jstack`. Jeśli adresy blokad w śladach stosu są różne, znaczy to, że reprezentują różne blokady. Innymi słowy, ślady stosu wątków, które mają inne adresy blokad, dotyczą wątków nierywalizujących o tę samą blokadę.

Kluczem do odnalezienia w listingu dla `jstack` blokad, o które toczy się rywalizacja, jest poszukiwanie tego samego adresu blokady dla kilku śladów stosu i odnalezienie wątków oczekujących na nabycie tego samego adresu blokady. Jeśli kilka śladów stosu wątków próbuje nabyć ten sam adres blokady, oznacza to, że w aplikacji zachodzi rywalizacja o blokady. Jeśli kilka listingów dla polecenia `jstack` daje podobne rezultaty w postaci obserwowanej rywalizacji o tę samą blokadę, jest to silne wskazanie, że w aplikacji istnieje blokada charakteryzująca się wysokim poziomem rywalizacji wśród wątków. Zwróć też uwagę, że ślad stosu podaje lokalizację kodu źródłowego takiej blokady aplikacji Java, a patrząc z perspektywy historycznej, uzyskanie tej informacji było zawsze dość trudnym zadaniem. Korzystanie z komendy `jstack` w sposób opisany powyżej może znacznie ułatwić odnajdywanie w aplikacji blokad, o które toczy się rywalizacja.

Bibliografia

„Monitoring and Management for the Java Platform”,
http://download.oracle.com/javase/1.5.0/docs/guide/management/.

„Java SE Monitoring and Management Guide”,
http://download.oracle.com/javase/6/docs/technotes/guides/management/toc.html.

„Connecting to JMX Agents Explicitly”,
http://download.oracle.com/javase/6/docs/technotes/guides/visualvm/jmx_connections.html.

VisualVM, *https://visualvm.dev.java.net/features.html*.

jvmstat 3.0, *http://java.sun.com/performance/jvmstat*.

Kup książkę Poleć książkę

Skorowidz

A

adaptacyjne ustalanie rozmiaru, 115, 269
agregacja plików dziennika, 405
akcja
  dołączania, 387
  jsp:useBean, 389
aktualizacja rekordu podatnika, 218
aktualizacje zbiorcze, 488
aktywności dysku, 359
alokacje
  char[], 224
  obiektów, 254
  pamięci, 89
  rejestru, 105
  struktur HashMap, 227
  w stosie wątku, 293
alokator obiektów, 97
analiza
  hierarchii klas, 105
  metryk blokad, 213
  skalowalności, 343
  ucieczki, 292
  wydajności
    bottom up, 29
    top down, 29
  wyników, 217
  zrzutu sterty, 201
  zrzutu wątku, 360
analizator, analyzer, 159
API wywołania, Invocation API, 90
aplikacja
  GlassFish, 347
  Java RMI server, 139
  jstatd daemon, 138
  NetBeans Profiler, 158, 185, 362
  Oracle Solaris Performance Analyzer, 158
  Oracle Solaris Studio Performance Analyzer, 159
  Performance Analyzer, 162–176, 212, 217–234
  VisualVM, 186
aplikacje
  enterprise, 352, 355, 362
  internetowe, 365
  klient–serwer, 62
  o niskich wymaganiach, 291
  wielowarstwowe, 325
architektura
  HotSpot VM, 72
  NUMA, 291
atrybut
  beanName, 391
  format, 403
  relative-catalogs, 417
  scope, 391
  transakcji, 467

B

badanie zmian wydajności, 342
bariera
  pamięci, 222
  zapisu, write barrier, 95
beany
  encyjne, 477
  enterprise, 468
  sesyjne, 451
    bezstanowe, 452
    stanowe, 452
  sterowane komunikatami, 451
benchmark, 298, 326
  asynchroniczny, 346
  prawo Little'a, 339
  skalowanie, 337
  typu enterprise, 328
  usługi internetowej, 425, 427
  uruchamianie, 343
  wielowarstwowy, 344
benchmarki dla EJB, 466
bezpieczeństwo, 139

Kup książkę Poleć książkę

bezpieczeństwo typów, 79
bezpośrednie wywoływanie procesów, 281
białe znaki, whitespaces, 388
biblioteka
  JSP, 392
  NIO, 369
biblioteki malloc, 399
binarne zrzuty sterty, 144, 202
blokada, 55
  JVM-System, 215
  przeciągana, biased locking, 85, 293
blokady problematyczne, 58
blokowanie, locking, 51
  bazy danych, 474, 489
  optymistyczne, 465, 474
  pesymistyczne, 474
błąd
  drugiego rodzaju, 322
  krytyczny, 91
  OutOfMemoryError, 92
  pierwszego rodzaju, 322
  segmentacji, segmentation fault, 91
  VerifyError, 81
Bootstrap Class Loader, 79
buforowanie
  danych, 207
  interfejsu Handle, 468
  pamięci, 395
  plików, 401
  referencji, 468
  schematów, 414

C

CAS, 219
charakterystyki
  aplikacji, 326–328
  wydajności, 430
cienki klient, 426
CLI, Command Line Interface, 350
CMP, container-managed persistence, 472
CMS, 121–125, 277, 284
cofanie transakcji, 474
CPI, cycles per instruction, 37, 204
CPU, 37, 204
CPU scheduler, 48
cykl
  CMS, 125, 278–280
  CPU, 36
  przetwarzania XML, 408
  zegara, 56
  życia maszyny wirtualnej, 75
  życia serwletu, 386
czas
  ekskluzywny, 160, 161
  inkluzywny, 160, 161
  ładowania fabryki, 410
  ładowania stron, 367
  na zastanowienie, 331, 333, 340
  odpowiedzi, response time, 335, 427
  pełnego przebiegu, 333
  procesora, 40
  przebiegu aplikacji, 211
  przestojów, 98, 130, 260
  przestojów CMS, 282
  reakcji, 242, 259
  równoczesny aplikacji, 128
  stabilności, 345
  trwania procesu, 262, 283
  uprzywilejowany, 39
  uruchamiania, startup time, 237, 242
  użytkownika, 39
  własny, 160
  zatrzymania, 128
  zmniejszania obciążenia, 345
  zwiększania obciążenia, 345
częstotliwość
  opóźnień, 260
  procesów, 262
  próbkowania, 188
  przestojów, 260
czyszczenie wstępne, pre-cleaning, 100

D

DAO, Data Access Objects, 471
debugger dbx, 91
debugowanie HotSpot VM, 309
definicja
  filtru, 177
  Java Collections, 226
  schematu, 442
delegowanie ładowarki klas, 79
deoptymalizacja, 106, 311, 315
długość
  odpowiedzi, 385
  ścieżki, path length, 30
dodatek
  Firebug, 331
  Live HTTP Headers, 331
dodawanie liczników, 49
dołączanie, include, 387
DOM, 412
domyślne wartości ergonomiczne, 113
domyślny
  mechanizm odzyskiwania, 112, 114
  współczynnik zapełnienia, 228

Kup książkę Poleć książkę

dostęp
do ziarna, 391
gruboziarnisty, 471
dostępność, availability, 241
driver framework, 337
drzewo wywołań, call tree, 159, 170
dyrektywa
dołączania, 387
usuwania, 388
dysk, 359
logiczny, 63
SDD, 360
dystrybucja zatrudnienia, 125, 276
dziedziczenie, 490
dziennik
błędów, 91
odzyskiwania pamięci, 257

E

EclipseLink, 464
eden, 147, 269
EJB, Enterprise Java Beans, 451
EJB 3.0, 478
EJB QL, 476
eksplozja obiektu, 293
EL, expression language, 391
element
any, 432
Summary, 419
eliminacja
barier odczytu, 293
sprawdzania zakresu, 104
synchronizacji, 293
wspólnych podwyrażeń, 104
encje
trwałe, 451
tylko do odczytu, 477
zewnętrzne, 415
ergonomiczne wartości domyślne, 111
estymacja
różnicy średnich, 320
średniej, 320
etapy profilowania, 161

F

fabryka abstrakcyjna, 409
Fast Infoset, 446, 448
faza
inicjacji, 240
podsumowująca, 240
ponownego zaznaczania, 282
równoczesnego zamiatania, 101
równoczesnego zaznaczania, 100
znaku inicjującego, 282
zrównoważona, 240
fazy ładowania klas, 78
FetchType, 485
FIFO, first in first out, 462
filtr, filter, 159
kompresji, 395
serwletu, 404
filtrowanie, filtering, 177
danych profilu, 177
zakresu próbki, 234
format
czasu, 249
daty, 249
fragmentacja, 101, 268
funkcja
ergonomii, 111
File Cache, 401
funkcje VisualVM, 137

G

głębokość kolejki, 42, 48
gniazdo połączenia, 355
GNOME System Monitor, 43
gorące miejsce, hot spot, 160
gruby klient, 426

H

HashMap, 214, 228
hierarchia klas, 105
hipoteza zerowa, 321
Historia CPU, 41
host wirtualny, 367
HotSpot debug VM, 309
HotSpot VM, 17, 55, 71
HotSpot VM Runtime, 72
HTTP service, 373
hybrydowy model skalowania, 343

I

identyfikacja zmian rozmiaru, 227
ilościowe okno próbkowania, 382
implementacja
beanu, 452
JAX-WS, 423
mikrobenchmarku, 312
punktu końcowego, 433
inicjowanie klas, 78

Kup książkę Poleć książkę

instalacja
NetBeans Profiler, 186
Oracle Solaris Studio Performance Analyzer, 162
instrukcja
ładowania, 204
wstępnego pobrania, 204
instrumentalizacja, instrumentation, 160
inteligentne wstawianie, 105
interakcja dysku, 65
interakcja z użytkownikiem, 329
interceptory metod biznesowych, 479
interfejs
Analyzera, 168
javax.ejb.Handle, 469
JNI, 90
narzędziowy JVMTI, 161
Provider, 444, 445
Shape, 311
Unmarshaller, 411
wiersza poleceń, 350
interfejsy
API, 418, 420
lokalne, 478
zdalne, 478
interpreter, 82
interwał pomiarowy, 298
IPC, instructions per clock, 37
IR, intermediate representation, 104
izolacja transakcji, 465
izolowanie
gorących blokad, 58
systemu testowego, 344
wycieków pamięci, 201

J

Java Collections, 226
Java Enterprise Edition, 325
Java Native Interface, 90
Java NIO, 211
Java SE, 220
JAXP, Java API for XML Binding, 408
JAX-WS RI, 423
jądro, kernel, 37
JDBC, Java Database Connectivity, 473
język
WSDL, 407, 430
wyrażeń EL, 391, 393
zapytań EJB, 475
JMX, Java Management eXtensions, 349
JNDI, Java Naming and Directory Interface, 459
JNI, Java Native Interface, 76, 90
JPA, Java Persistence API, 453
JSTL, JSP Standard Tag Library, 392
JVM, Java Virtual Machine, 71, 117, 237
JVMTI, JVM Tool Interface, 161

K

karty, cards, 94
katalogi XML, 415
klasa
Arena, 89
HashMap, 214, 216
StringBuffer, 223
StringBuilder, 223
klasy
implementacji fabryki parsera, 410
Java Collections, 226
rdzenia Java, 188
klienty JAX-WS, 426, 448
kod
fast-path, 85
obsługi błędów, 92
kodowanie
binarne, 446
wiadomości, 437
kody odpowiedzi, 378
kolorowanie wykresu, 110
komenda
asadmin, 351
asadmin set, 403
collect, 163, 362
csingle, 182
get, 352
iostat, 63
javaws, 75
JDK jstack, 154
jstack, 48, 156
ping, 354
typeperf, 41
vmstat, 45, 53
kompilacja warstwowa, 110
kompilacje OSR, 106
kompilator
javac, 80
JIT, 72, 83, 104, 150
kompilatory Javy, 80
kompresja
GZIP, 394
HTTP, 394, 448
w locie, 394
kompresor LZF, 399
konektor, 367
blokujący, 368
HTTP, 368

Kup książkę Poleć książkę

konfiguracja JMX, 142
konfigurowanie opcji zdalnego profilowania, 189
kontekst, 55
kontener
  EJB 3.0, 479
  webowy, 367, 373
konwersja dokumentów XML, 408
korporacja SPEC, 297
koszt
  instancjonowania ziarna, 390
  ładowania klasy, 81
  transmisji sieciowej, 397
  wykorzystania pamięci, 82
krótsza ścieżka, 203

L

launcher, 75
leniwe ładowanie, 472
liczba
  aktywnych beanów, 462
  instrukcji SQL, 488
  iteracji, 304, 313
  obserwacji, 319
  obsłużonych żądań, 380
  ocalałych bajtów, 274, 288
  optymalizacji pętli, 109
  procesorów wirtualnych, 111
  procesów, 130
  użytkowników, 339
  wątków, 290, 374
  wątków ORB, 487
  wirtualnych procesorów, 41
  załadowanych klas, 152
licznik
  backedge, 106
  wydajności, 49
  wywołań, invocation counter, 106
lokalizacja pamięci, 219
LRU, least recently used, 462
lustro Javy, Java mirror, 80

Ł

ładowanie, loading, 78
  klas, 78, 151
  węzła, 418
ładowarka bootstrap, 79
łańcuch Markowa, 330, 366
łączenie, linking, 78
łączenie w łańcuch, chaining, 207

M

macierz dyskowa, 360
magazyn danych, 222, 228
mapa tożsamości
  miękka, 455
  pełna, 455
  słaba, 455
mapowanie
  relacyjnej bazy danych, 453
  schematów, 432
  schematów XML, 431
  typów Javy, 441
martwy kod, 308
maszyna wirtualna, 71
  HotSpot, 252
  Javy, 212, 237, 352
MBeany, 153, 349, 386
MBeany GlassFish, 154
mechanizm
  CMS, 124, 284
  dołączania JSP, 387
  kopiujący, 96
  MTOM, 441
  odzyskiwania pamięci, 72, 93, 103, 246, 372
  przepustowościowy, 269, 285
  wykluczania beanów, 481
menedżer
  bezpieczeństwa, 371
  transakcji, 359
Menedżer Zadań, 38
metadane, 107
metadane klas w HotSpot, 80
metoda
  addEntry(), 201
  area(), 311
  checkResult(), 207
  close(), 208
  DestroyJavaVM, 77
  doTest(), 303, 304
  expandCapacity(), 224
  FileOutputStream.write(), 208
  flush(), 208
  HashMap.get(), 215, 220
  init, 386
  invoke(), 444
  JNI_CreateJavaVM, 76
  jspInit(), 387
  pass-by-reference, 471, 478
  pass-by-value, 470
  Random.next(), 217, 218
  reset(), 207
  service, 369

Kup książkę Poleć książkę

metoda
  stop-the-world, 151
  SynchronizedMap.get(), 215
  System.currentTimeMillis(), 301
  System.gc(), 129, 280, 372
  System.nanoTime(), 301
  write(), 207
  źródłowa, root method, 160
metody
  interceptora
    poziom domyślny, 479
    poziom klasy, 480
    poziom metody, 480
  kompilacji, 83
  natywne, 90
  poszukiwacza, 460
  statystyczne, 318, 323
metryka, 180
  sortowania, 180
  System CPU, 175
  User CPU, 175
  User Lock, 175
metryki
  pamięci, 136
  przypisane, 175
  wydajności, 333
mierzenie czasu, 302
migracja wątku, thread migration, 51, 59
mikrobenchmark, 297, 307, 315
minimalizator pliku, 389
młode pokolenie, 93, 95
model
  DOM, 412
  interakcji z użytkownikiem, 330
  pamięci Java, 221
  persystencji, 472
  wątkowania, 86
  wdrożenia
    pojedynczej instancji JVM, 243
    wielu instancji JVM, 243
modyfikacja programu, 221
Monitor
  Javy, 84
  Systemu, 41
  Systemu GNOME, 41, 42
  wydajności, 37, 39
monitoring zdalny, remote monitoring, 134
monitorowanie
  aktywności kompilacji JIT, 150
  aktywności ładowania klas, 152
  aplikacji Java, 153
  danych aplikacji, 386
  JVM, 117
  kolejki, 49, 50, 51
  kontenera EJB, 462
  ładowania klas, 152
  migracji wątków, 59
  mimowolnego przełączania kontekstu, 58
  podsystemów, 352
  progu zatrudnienia, 272
  puli połączeń, 361
  rywalizacji o blokady, 55, 154
  serwera aplikacji, 347
  we/wy sieci, 60
  wydajności, performance monitoring, 36, 118, 157
  wykorzystania CPU, 40
  wykorzystania pamięci, 52
  zasobów, 345
  zdalnych aplikacji, 138, 140
monitory Java/blokady, 213
MTOM, Message Transmission Optimization Mechanism, 436, 441

N

narzędzia graficzne, 132
narzędzie
  Btrace, 357
  collect, 159
  Collector, 362
  corestat, 68
  cpubar, 42
  cpustat, 66
  cputrack, 66
  er_print, 160, 166, 178–183, 362
  GCHisto, 129
  iobar, 64
  iostat, 63
  JConsole, 150, 132–134, 349
  JMeter, 330
  jstatd daemon, 139
  kstat, 66
  mpstat, 45, 214
  netstat, 59
  prstat, 46
  sar, 66
  Test TCP, 354
  top, 46
  typeperf, 50
  VisualVM, 137, 140, 144, 150
  vmstat, 45
nasłuchiwacz HTTP, 373
nasłuchiwacze, listeners, 369
nasłuchiwacze kontekstu, 387
nazwy metryk, 179
nieudana promocja, promotion failure, 97

Kup książkę Poleć książkę

niewydolność trybu równoczesnego, 126
nowa wartość, 219
NRU, not recently used, 462
NTP, Network Time Protocol, 346
NUMA, 291

O

obciążenie, overhead, 159
obiekt SchemaFactory, 414
obliczanie
  odchylenia standardowego, 319
  rozmiarów, 258
  średniej, 318
obsługa
  alokacji obiektów, 254
  błędów, 383
  błędów krytycznych VM, 91
  dynamicznych modyfikacji, 369
  wyjątków, 84
  załączników, 439
obszar wymiany, swap space, 51
odchylenie standardowe, 319, 321
odtwarzanie logów dostępu, 366
odwołanie blokady przeciąganej, 88
odzyskiwanie pamięci, 52, 88, 96, 129, 160, 246, 254, 266, 286, 300
  CMS, 277
  częstotliwość procesów, 259
  dla stałego pokolenia, 151
  dziennik, 257
  HotSpot VM, 93
  mark-compact, 98
  mechanizm
    pokoleniowy, 93
    przeważnie-równoczesny, 99
    równoczesny, 122
    równoległy, 99
    starego pokolenia, 104
    szeregowy, 98
    Train GC, 103
  mniejsze, 119, 265
  monitorowanie, 353
  monitorowanie procesów, 129
  najpierw-kosz, 102
  najpierw-śmieci, 102
  pełne, 119, 122, 265
  raportowanie danych, 119
  rozproszone, 372
  równoczesne, 281
  statystyki procesu, 119
  stop-the-world, 98
  szybka alokacja, 97
  typu stop-the-world, 270
  ustawienia HotSpot VM, 111
  VisualGC, 145
okno
  Attach Wizard, 188
  Filter Data, 178
  interfejsu Analyzer, 167
  ładowania próbki, 167
  Manual Integration, 190
  MBeans, 153
  Monitor, 141
  Overview, 140
  Profiling Results, 198
  Sampler, 143
  Set Data Presentation, 205
  Threads, 143
  VisualGC, 146
okres rozgrzewki, 298, 305, 307
oopsy, 73
opcja -Xloggc, 127
opcje
  komendy collect, 163
  wiersza poleceń, 74, 252, 292
  wiersza poleceń HotSpot VM, 248–250, 493–510
operacje
  CAS, 219
  punktów bezpieczeństwa, 250
  we/wy dysku, 359
  we/wy sieci, 354, 361
  wielowątkowe, 233
opóźnienie, latency, 237, 259, 354
optymalizacja, 105, 225, 292, 302
  agresywna, 315
  blokowania, 55
  CHA, 107
optymalizacje wysokiego poziomu, 110

P

paginacja, 483
pakiet
  Apache XML Commons Resolver, 415
  NetBeas IDE, 187
  sysstat, 59
  zdalnego profilowania, 190
pamięć, 51
pamięć podręczna, caching, 396
  beanów, 459
  drugiego poziomu, 453
  gotowa, 461
  kodu, 136
  L2, 464
  miękka, 455

Kup książkę Poleć książkę

pamięć podręczna, caching
  sesji JPA, 454
  transakcyjna, 461
  twarda, 455
  wyników, 454
  wyników zapytań, 484
panel
  Analyze Memory, 197
  Applications, 138
  Graphs, 148
  Histogram, 149
  kontrolny
    sekcja Basic Telemetry, 194
    sekcja Controls, 193
    sekcja Profiling Results, 194
    sekcja Status, 193
    sekcja View, 194
  programu profilującego, 192
  Spaces, 147
parser
  SAX, 419
  StAX, 410, 421
  Woodstox, 410
parsery strumieniowe, 412, 423
parsowanie dokumentu, 409
partycjonowanie żądań, 458
pełne odzyskiwanie pamięci, 97
percentyl, 335
pętla
  główna, 110
  instrukcji przełącznika, 83
  kończąca, 110
  wstępna, 110
planista krótkoterminowy
  w Linux, 51
  w Solaris, 50
  w Windows, 49
planista procesora, 48
plik
  default-web.xml, 389
  dziennika, 118
  ikony, 379
  jar, 81
  orm.xml, 484
  persistence.xml, 454, 487
  sun-cmp-mappings.xml, 473
  sun-ejb-jar.xml, 462, 471
  web.xml, 380
pływające śmieci, floating garbage, 101
pobieranie
  gorliwe, 473
  wstępne, prefetching, 473
podgląd strony, 333
podłączanie
  JConsole, 349
  VisualVM, 350
podprzestrzenie, subspaces, 82
podział iteracji, iteration splitting, 109
POJO, Plain Old Java Object, 452
pola statyczne, static fields, 78
polecenie, *Patrz* komenda
polityka
  bezpieczeństwa, security policy, 371
  wysiedlania, eviction policy, 462
połączenie JMX, 349
ponowne zaznaczanie, remark, 100
poprawa wydajności, 203, 206, 211
porównanie czasów, 312
POX, Plain Old XML, 446
poziomy izolacji transakcji, 465
prawdopodobieństwo wystąpienia błędu, 322
prawo Little'a, 339
prezentacja danych, 174
prędkość sieci, network speed, 354
proces profilowania, 143
proces rozwoju oprogramowania, 26, 27
procesory
  SPARC, 32, 67
  wirtualne, 41
procesy odzyskiwania pamięci, 129, 266, 286
profil, profile, 159
profilowanie, 143
  aplikacji enterprise, 362
  metod, 161, 186, 188
  pamięci, memory profiling, 157, 185, 197
  sterty, heap profiling, 157
  wydajności, performance profiling, 36, 118, 157
  wydajności CPU, 185
  z niskim obciążeniem, 185
program, *Patrz także* aplikacja
  obsługi, handlers, 434
  obsługi danych, 439
  obsługi JAX-WS, 427
  profilujący, profiler, 158
  rozwiązujący encje, 415
  rozwiązujący katalog, 416
projektowanie
  eksperymentów, 317
  testu, 338
promowanie, promotion, 248
proste transformacje tożsamości, 104
protokół SSL, 134
próba, experiment, 159
próg zatrudnienia, tenuring threshold, 125, 269–275
przebieg
  aplikacji, 298
  benchmarku, 335

Kup książkę Poleć książkę

przechwycenie profilu, 165, 223
przedwczesna promocja, premature promotion, 96
przedział ufności, 319–322
przekierowanie, 378
przełączanie kontekstu, context switching, 51
  dobrowolne, 55
  mimowolne, 55, 58
przepełnienie przestrzeni ocalałych, 270
przeploty, 84
przepływ
  pracy, 238
  sterowania, 106
przepustowość, throughput, 237, 241, 335, 354, 427
przepustowość sieci, network bandwidth, 59
przestój, stall, 37
przestrzenie nazw, 135
przestrzeń
  eden, 248
  młodego pokolenia, 248, 289
  ocalałych, 135, 147, 248, 269
  stałego pokolenia, 80, 151, 248
  starego pokolenia, 135, 148, 248
przesuwanie wskaźnika, 97
przetwarzanie
  binarnego bloku danych, 436
  dokumentów, 412
  dokumentów częściowe, 417
  dokumentu XML, 409
  strumieniowe, 368
  załączników, 439
  żądań, 377, 404
pule
  beanów, 459
  obiektów Unmarshaller, 411
  wątków, 457
  zasobów, 361
punkt końcowy EJB, 433
punkty bezpieczeństwa, safepoints, 80, 88
punkty bezpieczeństwa VM, 88

Q

QL, Query Language, 475

R

raportowanie metryki, 183
redukcja
  czasu uruchamiania, 82
  interakcji dysku, 65
  interwału próbkowania, 189
  przestojów CPU, 68
  użycia CPU, 62
refaktoryzacja, 230–232
referencje do interfejsu lokalnego, 470
regulacja
  adaptacyjna, 111
  czasu przestojów CMS, 282
  JVM, 237, 240, 260, 371
  keep-alive, 377
  kolejki połączeń, 376
  kontenera EJB, 456
  liczby wątków, 290
  maszyny wirtualnej, 353
  mechanizmu CMS, 277
  opóźnień, 259, 267
  pamięci podręcznej, 462
  przepustowości, 283
  przepustowości dla CMS, 284
  przepustowościowego mechanizmu, 285
  puli zasobów, 361
  puli wątków, 374, 458
  rozmiaru przestrzeni ocalałych, 287
  wydajności, performance tuning, 36, 118, 157
  wydajności kontenera webowego, 369
rejestrowanie zdarzeń dostępu, 402
replikacja pamięci wewnętrznej, 400
reprezentacja pośrednia, 104, 108
rezultaty bieżące, 198
rozgrzewka, 298, 307
rozkład t, 320
rozkładanie obciążenia, 283
rozlanie rejestru, register spilling, 73
rozlewanie wartości, 105
rozmiar
  bazy danych, 338
  bloku danych, 429
  edenu, 275
  HashMap, 228
  klas, 223
  pamięci podręcznej, 456, 464
  próbki, 319, 323
  przestrzeni
    młodego pokolenia, 115, 258
    ocalałych, 270, 274, 287, 288
    stałego pokolenia, 151, 258
    starego pokolenia, 263
  puli, 361, 463
    połączeń, 487
    wątków, 373
  sterty, 115, 254, 257
  sterty Java, 265
  stron pamięci, 295
  struktur danych, 222
  wiadomości, 428
  załącznika, 439
  żywych danych, 256

Kup książkę Poleć książkę

rozwiązywanie katalogu, 415
rozwijanie, unrolling, 104
równoczesny reset, concurrent reset, 125
rurociąg, 367
rywalizacja o blokady, 212
  w Linux, 57
  w Solaris, 55
  w Windows, 57
rzadkie pułapki, 107

S

serializacja, 397, 413
serializacja kontroli, 467
serializator
  Fast Infoset, 447
  Jackson, 399
serwer
  aplikacji, 347, 362
  GlassFish, 347, 368, 394, 458
  HTTP, 401
  StreamingDataHandler, 440
  WWW, 362
sesja EclipseLink, 464
sesja profilowania pamięci, 196
sieć, 354
silnik, engine, 367
silnik serwletu, 369
skalowalność, 59
skalowanie benchmarku, 337
skanowanie liniowe rejestru, 105
skryplety, 393
skrypt iosnoop.d, 64
skrypty powłoki, 362
słabe referencje, 455
słowo cechujące, mark word, 85
słowo kluczowe
  transient, 400
  volatile, 221
SMP, symmetric multiprocessing, 343
SOA, Service Oriented Architecture, 407
Solaris cpubar, 44
spadek wydajności, 73, 314
specyfikacja
  EJB, 451
  Fast Infoset, 447
  Java Persistence, 490
  JAX-WS, 423
  języka Java, 78
  JPA, 453, 482
  SOAP, 446
SSA, static single assignment, 104, 108
stała czasowa, constant time, 85
stałe pokolenie, 94
stany wątku, 87
stara wartość, 219
stare pokolenie, 93
statyczne inicjalizatory klas, 78
statystyki
  blokad, 214
  odzyskiwania pamięci, 127
  przestojów, 130
  serwera aplikacji, 350
  wydajności, 38, 380
sterownik
  HTTP, 427
  testowy, 338
sterta, heap, 89, 160, 252
stos
  JAX-WS RI, 424
  wywołań, 173
strategia alokacji rejestru, 105
stronicowanie, paging, 42, 51
strony pamięci, 294
struktury HashMap, 227
supersłowo, superword, 110
SUT, system under test, 328, 426
symulacja opóźnień, 332
synchronizacja, 84
system
  Solaris, 42
  testowy SUT, 328
  zewnętrzny, 356
szablon, template, 82
szerokość pasma, bandwidth, 354
szybkość skanowania pamięci, 42

Ś

ścieżka
  klasy, 79
  wykonania, 107, 201
ślady stosu, 92, 155
śledzenie
  stosu, stack trace, 48
  żądania, 404
średni czas odpowiedzi, 335
średnia, 318
środowisko Java Runtime, 81
środowisko uruchomieniowe JVM, 244

T

tablica kart, card table, 94
tablice, arrays, 82
tablice bazy danych, 490
technologia Fast Infoset, 447

Kup książkę Poleć książkę

test
obciążenia, load test, 166
t-Studenta, 321
testowanie, 240
testy
porównawcze, 297, 325, 347, 366
porównawcze usług internetowych, 425
wydajności, 216
TLAB, Thread-Local Allocation Buffers, 97
transakcja
kupuj, 336
użytkownika, 333
wyszukiwanie, 336
transakcje zarządzane przez
beany, 466
kontener, 466
trwałość sesji, 400
tryb
deweloperski, 370
eksperta, 176
maszyny, 176
podłączania, 188
produkcyjny, 370
stop-the-world, 267
użytkownika, 176
tryby prezentacji danych, 176
tworzenie
benchmarku, 331, 332
instancji Factory, 410
klienta proxy, 449
klienta usługi internetowej, 426
mikrobenchmarków, 329
parsera, 411
puli połączeń, 486
wątku, 86
typ
pamięci podręcznej, 456
pobierania
FetchType, 485
EAGER, 485
LAZY, 485
typy schematów XML, 432

U

udostępnianie danych klas, 79, 81
ufność, 322
ujemny rozkład wykładniczy, 332
układ sterty, 252
unieważnienie pamięci podręcznej, 484
unmarshaller JAXB, 412
urządzenia wejścia/wyjścia, 37
usługi
HTTP, 373
internetowe, 407, 425, 427
usunięcie poza pętlę, 105
usuwanie
białych znaków, 388
martwego kodu, 302
sprawdzania zakresów, 110
użycie pamięci, memory footprint, 242

V

VisualGC GUI, 145

W

wątek, 85
Blocked thread, 87
New thread, 87
Thread in Java, 87
Thread in vm, 87
wątki
natywne, 86
wewnętrzne VM, 87
wdrożenie JVM, 243
wejście w monitor, 84
wersje JVM, 245
weryfikacja, validation, 338, 413
kodu bajtowego, 80
typu, type veryfication, 81
wnioskowanie typu, type inference, 81
wplatanie, 307, 310
WSDL, Web Services Description Language, 407, 430
wskaźniki obiektów, 80
skompresowane, 73
typowe, 73
wspólne archiwum, 81
współczynnik
NewRatio, 372
promowania obiektów, 266
przybywania, 342
równoległości, 230
wstrzykiwania, 332
wstawianie treści, 104
wtyczka
VisualGC, 145
VisualVM-MBeans, 153
wybór interfejsu API, 420
wyciek pamięci, memory leak, 160, 201
wydajność, 203, 246, 314, 333
aplikacji, 325, 379
aplikacji internetowych, 365
buforowania, 469

Kup książkę Poleć książkę

wydajność
częściowego przetwarzania, 443
DOM i JAXB, 422
EJB QL, 477
interfejsów Provider, 446
Java Persistence, 451
JAX-WS, 445
klienta usługi internetowej, 449
metod, 307, 310
numerycznych typów schematów, 431
parserów strumieniowych, 421
parsowania, 414
pełnego przetwarzania, 443
pobierania, 474
procesu serializacji, 422
programu, 224, 434
punktów końcowych, 434
schematów, 432
serializatorów, 399
sieci, 354
skrypletu, 393
systemów zewnętrznych, 356
systemu, 33
usług internetowych, 407, 428
we/wy sieci, 59
XML, 408
wyjątki, exceptions, 84
wyjście z monitora, 84
wykorzystanie blokad, 175
wykorzystanie
CPU, 175, 205, 232, 336
przez system, 160
przez użytkownika, 160
pamięci
w Linux, 54
w Solaris, 53
w Windows, 52
sieci, 62
we/wy dysku, 63
we/wy sieci, 60
załączników, 443
wykres
Compile Time, 150
programowo zależny, 108
wykrywanie wycieków pamięci, 185
wymagania systemowe, 238, 241
wymiana stron, swapping, 51
wyścigi, races, 84
wyświetlanie
danych, 213
metod, 183
wywołujący-wywoływany, Caller-Callee, 160
wywoływanie procesów, 281
wzajemne wykluczenie, mutual exclusion, 84
wzrost wydajności, 220

X

XOP, XML-binary Optimized Packaging, 436

Z

zaczepy, hooks, 451
zakleszczenie wątków, thread deadlocks, 92
zakładka
Callers-Callees, 209, 215, 217
GC Timeline, 131
Memory, 135
Timeline, 233
Zasoby, 41
załączanie dokumentu XML, 441
załączniki, 443
zamiana skalara, 293
zapełnianie
przestrzeni ocalałych, 276
HashMap, 228
zapytania
dynamiczne, 482
języka zapytań JPA, 482
JPA QL, 483
natywne, 482
nazwane, 482
zarządzalność, manageability, 241
zarządzanie
bazą danych, 400
powtarzalnością, 345
wątkami, 85
zatrudnianie, tenuring, 248, 271
zawijanie, wrapping, 207
zbieranie danych profilu, 166
zdalne profilowanie, 144, 187
zdalny
host, 140, 190
monitoring, 142
system, 138
zdarzenia dostępu, 402
ziarnistość usługi, 429
zmiana rozmiaru, 224, 227
HashMap, 228
klas, 223, 226
StringBuilder, 229
zmienność, 323
czasu odpowiedzi, 341
przepustowości, 341
wykorzystania CPU, 341

Kup książkę Poleć książkę

znacznik daty i czasu, 126
znak inicjujący, initial mark, 100
zrzuty
  ekranu rezultatów, 196, 200
  sterty, 145, 201
  stosu wątku, 88
  wątków, threads dumps, 353
zsynchronizowane bloki, 84
zużycie pamięci, memory footprint, 237, 251
zwijanie stałych, 104

Ż

żądania HTTP, 404
żądanie, request, 333

Kup książkę Poleć książkę

Kup książkę Poleć książkę

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl

GRUPA WYDAWNICZA
Helion SA

Wydajność JAVY

Na rynku znajduje się już kolejne wydanie języka Java, oznaczone numerem 7. Oto najlepszy dowód, że język ten ma się dobrze i wciąż jest na topie. Oczywiście to potwierdza, że programiści Javy są jedną z najbardziej rozchwytywanych grup na rynku pracy. Dlatego warto zainwestować w naukę tego języka. Do Javy przylgnęła krzywdząca opinia, że jest powolna i mało wydajna, ale to mit! W dzisiejszych czasach język ten ani na krok nie ustępuje innym, a jeżeli zastosujesz się do wskazówek zawartych w tej wyjątkowej książce, może je nawet prześcignąć!

Na początku lektury dowiesz się, jak wiarygodnie monitorować obciążenie systemu operacyjnego — zużycie pamięci, obciążenie procesora oraz sieci. Następnie przejdziesz do tego, na co czekasz najbardziej: do dostrajania Twojej aplikacji oraz wirtualnej maszyny Java. Poznasz szczegółowo zasady działania mechanizmów odzyskujących pamięć oraz dostępne przełączniki, które potrafią w znaczący sposób wpłynąć na wydajność środowiska. Ponadto zgłębisz tematykę wydajności aplikacji internetowych oraz technologii EJB i JPA. Jest to długo oczekiwana na polskim rynku pozycja, poświęcona zagadnieniom niezwykle istotnym z punktu widzenia programisty. To Twoja obowiązkowa lektura na najbliższe dni!

Przekonaj się, jak szybka może być Java dzięki:

- wykorzystaniu właściwych przełączników
- zastosowaniu narzędzi do profilowania
- wyborowi właściwej wersji 32- lub 64-bitowej
- optymalizacji wykorzystania sieci i pamięci

Wyciśnij siódme poty z wirtualnej maszyny Java!

helion.pl
księgarnia internetowa

Nr katalogowy: 12162

Księgarnia internetowa:
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ
KOD KORZYŚCI

ISBN 978-83-246-4380-6



Cena: 89,00 zł