O'REILLY®

Wydanie II

Wydajność Javy

Szczegółowe porady dotyczące programowania i strojenia aplikacji w Javie

Helion

Scott Oaks

Tytuł oryginału: Java Performance: In-Depth Advice for Tuning and Programming Java 8, 11, and Beyond, 2nd Edition

Tłumaczenie: Andrzej Watrak

ISBN: 978-83-283-7031-9

Helion SA
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/wydja2.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/wydja2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 7.

Dobre praktyki używania sterty

W rozdziałach 5. i 6. szczegółowo opisałem, jak dostroić kolektor, aby jego wpływ na działanie aplikacji był jak najmniejszy. Strojenie kolektora to ważna operacja, ale zazwyczaj jeszcze lepszy efekt daje stosowanie dobrych praktyk programistycznych. W tym rozdziale opisuję kilka takich praktyk dotyczących wykorzystania sterty.

W tym kontekście pojawiają się dwa sprzeczne podejścia. Podstawową zasadą jest oszczędne korzystanie z obiektów i usuwanie ich tak szybko, jak jest to możliwe. Im mniej pamięci wykorzystuje aplikacja, tym efektywniej działa kolektor. Z drugiej strony, częste tworzenie i usuwanie obiektów pogarsza ogólną wydajność aplikacji, nawet jeżeli kolektor działa sprawniej. Jeżeli natomiast obiekty są używane wielokrotnie, aplikacja działa znacznie efektywniej. Obiekty można wielokrotnie wykorzystywać na kilka sposobów, m.in. stosując lokalne zmienne wątkowe, specjalne odwołania i pule obiektów. Tego rodzaju obiekty są długotrwałe i wpływają na działanie kolektora; umiejętnie wykorzystywane, poprawiają ogólną wydajność aplikacji.

W tym rozdziale opisuję różne podejścia i ich konsekwencje. Najpierw jednak przyjrzyjmy się kilku narzędziom, dzięki którym można się dowiedzieć, co się dzieje wewnątrz sterty.

Analiza sterty

Dziennik kolektora i narzędzia opisane w rozdziale 5. są doskonałymi źródłami informacji o wpływie porządkowania pamięci na działanie aplikacji. Jednak aby uzyskać jeszcze dokładniejszy obraz, trzeba zajrzeć do samej sterty. Narzędzia opisane w tym podrozdziale dostarczają informacji o obiektach aktualnie wykorzystywanych przez aplikację.

Większość narzędzi uwzględnia tylko aktywnie wykorzystywane obiekty. Te, które zostaną usunięte podczas najbliższego pełnego porządkowania pamięci, nie są uwzględniane w prezentowanych wynikach. W pewnych sytuacjach narzędzia te inicjują pełne porządkowanie pamięci, co może mieć wpływ na działanie aplikacji. Oprócz tego analizują całą stertę i zbierają informacje o wykorzystywanych obiektach, ale nie usuwają tych, które nie są już potrzebne. W obu przypadkach narzędzia wykorzystują zasoby komputera przez pewien czas, dlatego nie należy ich stosować do mierzenia wydajności aplikacji.

Histogram sterty

Ograniczanie zajętości pamięci jest bardzo ważne, ale tak jak w każdej kwestii dotyczącej wydajności, należy skupić się na maksymalizacji możliwych do uzyskania korzyści. W dalszej części rozdziału poznasz przykład leniwego inicjowania obiektu typu `Calendar`. Jest to operacja, dzięki której można zaoszczędzić 640 bajtów pamięci. Jednak jeżeli ten sam obiekt będzie inicjowany wielokrotnie, różnica w wydajności aplikacji będzie niezauważalna. Celem analizy sterty jest uzyskanie informacji o obiektach zajmujących duże ilości pamięci. Najłatwiej można to osiągnąć, tworząc **histogram sterty**. Jest to szybki sposób dający wgląd w liczbę obiektów bez konieczności zrzucania całej sterty (zrzut zajmuje sporo miejsca na dysku, a jego analiza trwa dłuższą chwilę). Jeżeli za brak pamięci są odpowiedzialne obiekty kilku określonych typów, z pomocą histogramu można je szybko zidentyfikować.

Histogram sterty można utworzyć za pomocą narzędzia `jcmd`. Ilustruje to poniższy przykład. Proces analizowanej aplikacji miał w tym przypadku identyfikator 8898:

```
% jcmd 8998 GC.class_histogram
8898:
 num     #instances         #bytes  class name
----------------------------------------------
   1:        789087       31563480  java.math.BigDecimal
   2:        172361       14548968  [C
   3:         13224       13857704  [B
   4:        184570        5906240  java.util.HashMap$Node
   5:         14848        4188296  [I
   6:        172720        4145280  java.lang.String
   7:         34217        3127184  [Ljava.util.HashMap$Node;
   8:         38555        2131640  [Ljava.lang.Object;
   9:         41753        2004144  java.util.HashMap
  10:         16213        1816472  java.lang.Class
```

Na początku histogramu zazwyczaj znajdują się informacje o tablicy znaków (`[C`) i obiektach typu `String`, ponieważ są to najczęściej wykorzystywane obiekty. Często pojawiają się też tablice bajtów (`[B`) i obiektów (`[Ljava.lang.Object;`), ponieważ w tego typu strukturach przechowują swoje dane obiekty ładujące klasy (ang. *classloaders*). Jeżeli zawartość histogramu nie jest dla Ciebie zrozumiała, zapoznaj się z dokumentacją do interfejsu JNI.

W tym przykładzie należy wyjaśnić obecność obiektów typu `BigDecimal`. Wiadomo, że aplikacja tworzy wiele krótkotrwałych obiektów tego typu, ale tak duża ich liczba jest czymś nietypowym. Histogram zawiera tylko obiekty wykorzystywane w aplikacji, ponieważ polecenie użyte do jego utworzenia wymusza pełne porządkowanie pamięci. Jeżeli użyje się parametru `-all`, porządkowanie nie zostanie wykonane i histogram będzie uwzględniał wszystkie obiekty, również te niepotrzebne (bez odwołań).

Podobny wynik można uzyskać za pomocą następującego polecenia:

```
% jmap -histo identyfikator_procesu
```

Wynik powyższego polecenia uwzględnia obiekty przeznaczone do usunięcia. Aby wymusić pełne porządkowanie pamięci, należy użyć następującego polecenia:

```
% jmap -histo:live identyfikator_procesu
```

Histogram jest prostym raportem, więc warto go tworzyć w zautomatyzowanym systemie testowania aplikacji. Nie należy go jednak generować podczas mierzenia wydajności ustabilizowanej aplikacji, ponieważ zajmuje to kilka sekund, a ponadto inicjuje pełne porządkowanie pamięci.

Zrzut sterty

Histogram doskonale nadaje się do diagnozowania problemów wynikających z alokowania zbyt wielu instancji jednej lub kilku określonych klas. Jednak do przeprowadzenia głębszej analizy potrzebny jest **zrzut sterty**. Dostępnych jest wiele narzędzi do wykonywania tego rodzaju analiz. Większość z nich pozwala podłączyć się pod działającą aplikację i tworzyć zrzut. Najłatwiej robi się to za pomocą następujących poleceń:

```
% jcmd identyfikator_procesu GC.heap_dump /ścieżka/zrzut.hprof
```

lub

```
% jmap -dump:live,file=/ścieżka/zrzut.hprof identyfikator_procesu
```

Opcja `live` w powyższym poleceniu powoduje, że narzędzie przed zrzuceniem sterty przeprowadza jej pełne porządkowanie. Jest to domyślne działanie narzędzia. Jeżeli jednak z jakiegoś powodu zrzut powinien zawierać niewykorzystywane obiekty, należy na końcu polecenia użyć parametru `-all`. Narzędzie, wykonując pełne porządkowania pamięci, wprowadza oczywiście długą pauzę w działaniu aplikacji. Jednak nawet bez porządkowania aplikacja jest wstrzymywana na dość długi czas, niezbędny do zapisania sterty w pliku.

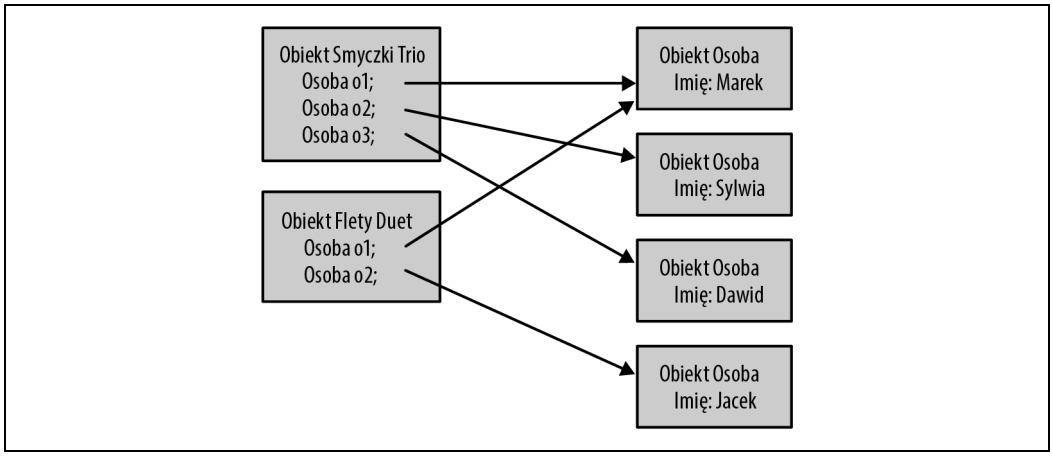
Oba powyższe polecenia tworzą plik o nazwie *zrzut.hprof* we wskazanym katalogu. Do analizowania zrzutu służy wiele różnych narzędzi. Poniżej opisane są najczęściej stosowane.

`jvisualvm`
: W zakładce *Monitor* powyższego narzędzia można zrzucić stertę działającej aplikacji lub otworzyć zrzut wygenerowany wcześniej. Oprócz tego można przeglądać zawartość sterty, wyszukiwać największe obiekty i przetwarzać stertę za pomocą specjalnych zapytań.

`mat`
: Do bezpłatnego narzędzia EclipseLink Memory Analyzer Tool (`mat`) można załadować jeden lub kilka zrzutów i analizować je. Narzędzie tworzy raporty zawierające informacje o źródłach potencjalnych problemów. Umożliwia też przeglądanie sterty za pomocą zapytań podobnych do SQL.

Analiza sterty obejmuje przede wszystkim zachowaną pamięć obiektów (ang. *retained memory*), tj. taką, która byłaby dostępna, gdyby można było te obiekty usunąć. Na rysunku 7.1 jest to pamięć zajmowana przez obiekt `Smyczki Trio`, jak również obiekty `Sylwia` i `Dawid`. Nie jest to jednak pamięć zajmowana przez obiekt `Marek`, ponieważ odwołuje się do niego inny obiekt. Dlatego nie można go usunąć podczas porządkowania pamięci wraz z obiektem `Smyczki Trio`.

Obiekty zajmujące duże ilości pamięci zachowanej są nazywane **dominatorami** sterty. Jeżeli narzędzie analityczne pokazuje, że sterta jest zdominowana przez kilka obiektów, problem jest prosty. Aby go rozwiązać, wystarczy zmniejszyć ich liczbę, używać ich przez krótszy czas, uprościć ich strukturę lub pomniejszyć je. Wprawdzie łatwiej to powiedzieć niż zrobić, ale przynajmniej analiza jest prosta.



Rysunek 7.1. Struktura zachowanej pamięci

Wielkości płytkich, zachowanych i głębokich obiektów

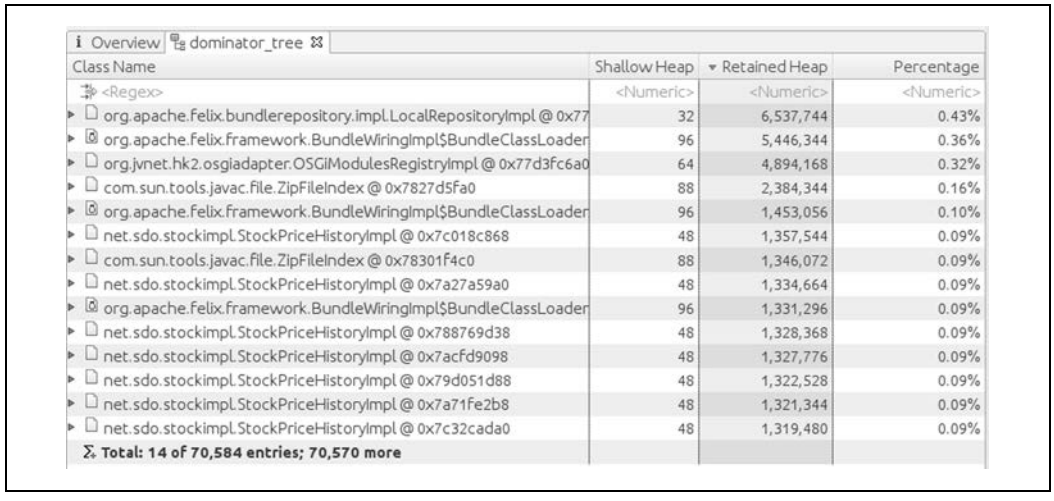
Dwa inne często stosowane w analizie sterty terminy to obiekty płytkie (ang. *shallow*) i głębokie (ang. *deep*). Określenie *wielkość obiektu płytkiego* oznacza wielkość samego obiektu. Jeżeli dany obiekt zawiera odwołania do innych obiektów, powyższy termin uwzględnia dodatkowe 4 bajty (lub 8 bajtów) zajmowane przez każde odwołanie, ale nie obejmuje wielkości obiektów docelowych.

Termin *wielkość głębokiego obiektu* obejmuje również wielkości obiektów, do których odwołuje się dany obiekt. Różnica pomiędzy wielkością głębokiego obiektu a zachowaną pamięcią to suma wielkości obiektów współdzielonych. Na rysunku 7.1 pamięć zajmowana przez obiekt `Marek` jest uwzględniona w wielkości głębokiego obiektu `Flety Duet`, ale nie w jego pamięci zachowanej.

Zazwyczaj jednak w celu zdiagnozowania problemu trzeba przeprowadzać dochodzenie detektywistyczne, ponieważ często obiekty są współdzielone. Wielkość tego rodzaju obiektów, jak np. `Marek` na powyższym rysunku, nie jest uwzględniana w zachowanej pamięci, ponieważ usunięcie nadrzędnego obiektu nie powoduje usunięcia obiektu współdzielonego. Ponadto najwięcej zachowanej pamięci zajmują obiekty ładujące klasy, nad którymi nie ma się praktycznie żadnej kontroli. Rysunek 7.2 przedstawia ekstremalny przypadek serwera danych giełdowych. Na początku listy znajdują się obiekty zajmujące najwięcej pamięci zachowanej. Serwer umieszcza obiekty z silnymi i słabymi odwołaniami odpowiednio w pamięci podręcznej i w globalnej tablicy mieszającej. Zatem w pamięci podręcznej znajdują się obiekty zawierające wiele odwołań.

Obiekty zajmują 1,4 GB sterty (ta wartość nie jest widoczna w narzędziu). Jednak największy zbiór obiektów posiadających jedno odwołanie ma wielkość tylko 6 MB (nie powinno dziwić, że jest to część platformy ładującej klasy). Przeglądanie obiektów zajmujących najwięcej pamięci zachowanej nie ułatwia diagnostyki problemów z pamięcią.

W tym przykładzie widocznych jest wiele instancji klasy `StockPriceHistoryImpl`, z których każda zajmuje sporo pamięci zachowanej. Na podstawie ilości pamięci zajmowanej przez te obiekty można wywnioskować, że tu leży problem. Jednak zazwyczaj obiekty mogą być współdzielone, więc analizując zachowaną pamięć, nie da się wyciągnąć jednoznacznych wniosków.



Overview | dominator_tree

Class Name	Shallow Heap	Retained Heap	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
org.apache.felix.bundlerepository.impl.LocalRepositoryImpl @ 0x77	32	6,537,744	0.43%
org.apache.felix.framework.BundleWiringImpl$BundleClassLoader	96	5,446,344	0.36%
org.jvnet.hk2.osgiadapter.OSGiModulesRegistryImpl @ 0x77d3fc6a0	64	4,894,168	0.32%
com.sun.tools.javac.file.ZipFileIndex @ 0x7827d5fa0	88	2,384,344	0.16%
org.apache.felix.framework.BundleWiringImpl$BundleClassLoader	96	1,453,056	0.10%
net.sdo.stockimpl.StockPriceHistoryImpl @ 0x7c018c868	48	1,357,544	0.09%
com.sun.tools.javac.file.ZipFileIndex @ 0x78301f4c0	88	1,346,072	0.09%
net.sdo.stockimpl.StockPriceHistoryImpl @ 0x7a27a59a0	48	1,334,664	0.09%
org.apache.felix.framework.BundleWiringImpl$BundleClassLoader	96	1,331,296	0.09%
net.sdo.stockimpl.StockPriceHistoryImpl @ 0x788769d38	48	1,328,368	0.09%
net.sdo.stockimpl.StockPriceHistoryImpl @ 0x7acfd9098	48	1,327,776	0.09%
net.sdo.stockimpl.StockPriceHistoryImpl @ 0x79d051d88	48	1,322,528	0.09%
net.sdo.stockimpl.StockPriceHistoryImpl @ 0x7a71fe2b8	48	1,321,344	0.09%
net.sdo.stockimpl.StockPriceHistoryImpl @ 0x7c32cada0	48	1,319,480	0.09%
Total: 14 of 70,584 entries; 70,570 more			

Rysunek 7.2. Widok pamięci zachowanej w narzędziu Memory Analyzer

Kolejnym ważnym krokiem może być utworzenie histogramu obiektów, przedstawionego na rysunku 7.3.

Overview | Histogram

Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
java.math.BigDecimal	12,920,067	516,802,680	517,429,776
java.util.TreeMap$Entry	7,255,390	290,215,600	1,450,796,576
net.sdo.stockimpl.StockPriceImpl	7,240,530	289,621,200	980,225,584
java.util.Date	7,244,268	173,862,432	174,077,552
net.sdo.stockimpl.StockPricePK	7,240,530	173,772,720	173,799,360
char[]	266,992	25,934,280	25,934,280
java.lang.String	255,336	6,128,064	30,780,696
java.util.HashMap$Entry[]	59,102	5,050,328	30,515,800
java.util.HashMap$Entry	151,237	4,839,584	30,295,176
java.util.LinkedHashMap$Entry	72,786	2,911,440	6,298,496
com.sun.tools.javac.file.ZipFileIndex$Entry	44,416	2,131,968	6,049,552
java.lang.Object[]	31,328	1,930,928	23,857,992
java.util.HashMap	34,114	1,910,384	29,772,824
java.lang.reflect.Method	21,579	1,726,320	3,714,040
Total: 14 of 12,007 entries; 11,993 more	**43,446,283**	**1,517,322,152**	

Rysunek 7.3. Histogram obiektów w narzędziu Memory Analyzer

Histogram zawiera podsumowanie informacji o obiektach tych samych typów. W tym przypadku widać wyraźnie, że źródłem problemu jest 7 milionów obiektów typu `TreeMap$Entry`, zajmujących przestrzeń o wielkości 1,4 GB. Za pomocą narzędzia Memory Analyzer można łatwo wyśledzić tego rodzaju obiekty i sprawdzić ich zawartość, nawet gdy nie wiadomo, co się dzieje wewnątrz aplikacji.

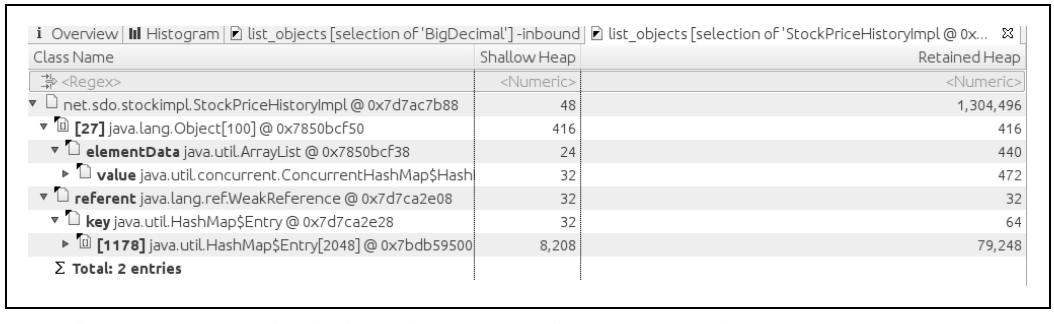
Za pomocą narzędzia do analizy sterty można łatwo znaleźć główne obiekty (lub ich zbiór, jak w tym przykładzie), od których zaczyna się porządkowanie pamięci. Jednak zazwyczaj nie jest to pomocna informacja. Obiekty główne zawierają statyczne, globalne odwołania do poszukiwanych obiektów, prowadzące zazwyczaj przez długi łańcuch innych obiektów. Najczęściej są to statyczne

zmienne w klasie znajdującej się w ścieżce systemowej lub ścieżce bootstrap. Jest to m.in. klasa `Thread` wraz ze wszystkimi aktywnymi wątkami. Wątki zapisują obiekty za pośrednictwem lokalnych zmiennych lub odwołań do docelowego obiektu `Runnable` (ewentualnie do klasy pochodnej od `Thread` i wszystkich zawartych w niej odwołań, jak w tym przykładzie).

Czasami informacja o głównych obiektach jest pomocna. Jednak jeżeli obiekt zawiera wiele odwołań, to jest połączony z wieloma obiektami głównymi. Odwołania tworzą strukturę odwróconego drzewa. Załóżmy, że dwa obiekty odwołują się do obiektu `TreeMap$Entry`. Do każdego z nich mogą odwoływać się dwa inne obiekty, z kolei do każdego z tych obiektów mogą odwoływać się trzy inne itd. Liczba odwołań, gwałtownie rosnąca w miarę zbliżania się obiektów głównych, oznacza, że do każdego obiektu odwołuje się wiele obiektów głównych.

Dlatego bardziej pomocne jest wcielenie się w detektywa i znalezienie współdzielonego obiektu znajdującego się na najniższym poziomie struktury. Można to osiągnąć analizując obiekty i odwołania do nich do momentu znalezienia powielonych ścieżek. W tym przykładzie do obiektów `Stock ↪PriceHistoryImpl` odwołują się dwa inne obiekty: `ConcurrentHashMap`, zawierający atrybuty sesji, oraz `WeakHashMap`, będący globalną pamięcią podręczną.

Na rysunku 7.4 rozwinięte są tylko początki ścieżek dwóch obiektów. Aby się przekonać, że zawierają one dane sesji, należy dalej rozwijać ścieżkę obiektu `ConcurrentHashMap`, aż stanie się to oczywiste. Podobnie należy potraktować obiekt `WeakHashMap`.



Rysunek 7.4. Wsteczne ścieżki odwołań widoczne w narzędziu Memory Analyzer

W tym przykładzie, z racji zastosowanych typów obiektów, analiza była nieco prostsza niż zazwyczaj bywa w praktyce. Jeżeli główne dane wykorzystywane w aplikacji są przechowywane w obiektach typu `String`, a nie `BigDecimal`, lub w strukturach typu `HashMap`, a nie `TreeMap`, wtedy zadanie staje się trudniejsze. Zrzut sterty może zawierać setki tysięcy obiektów typu `String` i `HashMap`, więc znalezienie tych właściwych jest żmudną czynnością. Zgodnie z niepisaną zasadą, należy wtedy zacząć od obiektów reprezentujących kolekcje, np. `HashMap`, a nie wejścia, jak `HashMap$Entry`, i poszukać największych kolekcji.

Krótkie podsumowanie

- Znalezienie obiektów zajmujących najwięcej pamięci jest pierwszym krokiem do optymalizacji kodu.
- Za pomocą histogramu można szybko i łatwo diagnozować problemy w pamięcią, których źródłem jest duża liczba obiektów określonego typu.
- Analiza zrzutu sterty jest jedną z najskuteczniejszych technik diagnozowania problemów z zajętością pamięci. Wymaga jednak cierpliwości i wysiłku.

Problem przepełnienia pamięci

Maszyna JVM zgłasza błąd *przepełnienia pamięci* w następujących przypadkach:

- braku wystarczającej ilości natywnej pamięci,
- zapełnienia metaprzestrzeni,
- zapełnienia sterty (aplikacja nie jest w stanie tworzyć nowych obiektów),
- zbyt długiego czasu porządkowania pamięci.

Najczęściej występują dwa ostatnie przypadki, które są związane ze stertą. Jednak przepełnienie pamięci nie oznacza od razu, że problem dotyczy sterty. W takiej sytuacji trzeba sprawdzić, co jest przyczyną błędu (zazwyczaj zawiera ją opis zgłoszonego wyjątku).

Brak wystarczającej ilości natywnej pamięci

Pierwsza wymieniona na powyższej liście przyczyna, brak wystarczającej ilości fizycznej pamięci, nie jest związana ze stertą. Maszyna JVM w wersji 32-bitowej jest w stanie obsłużyć pamięć o wielkości 4 GB (w starszych wersjach systemu Windows było to 3 GB, a Linux — 3,5 GB). Skonfigurowanie bardzo dużej sterty, np. 3,8 GB, powoduje niebezpieczne zbliżenie się do tej granicy. Nawet dla 64-bitowej maszyny JVM system operacyjny może nie mieć wystarczającej ilości wirtualnej pamięci.

Ten temat będzie dokładniej opisany w rozdziale 8. Należy jednak pamiętać, że gdy pojawi się komunikat o braku wystarczającej ilości natywnej pamięci, strojenie sterty nic nie da. Zamiast tego należy dokładniej przyjrzeć się komunikatowi. Na przykład poniższy informuje, że przepełniła się natywna pamięć przeznaczona dla stosu wątków:

```
Exception in thread "main" java.lang.OutOfMemoryError:
unable to create new native thread
```

Czasami zdarza się, że maszyna JVM zgłasza problem z powodu zupełnie niezwiązanego z natywną pamięcią. Aplikacje zazwyczaj mogą uruchamiać ograniczoną liczbę wątków, określoną przez system operacyjny lub kontener. Na przykład w systemie Linux aplikacja może utworzyć 1024 wątki (wartość tę można sprawdzić za pomocą polecenia `ulimit -u`). Przy próbie uruchomienia 1025. wątku pojawi się błąd `OutOfMemoryError` i wygenerowany zostanie komunikat o niewystarczającej ilości natywnej pamięci, choć rzeczywistą przyczyną problemu jest przekroczenie określonej przez system operacyjny liczby wątków.

Zapełnienie metaprzestrzeni

Błąd zapełnienia metaprzestrzeni również nie jest związany ze stertą. Pojawia się wtedy, gdy zapełni się natywna pamięć zajmowana przez metaprzestrzeń. Ponieważ domyślnie maksymalna wielkość metaprzestrzeni nie jest określona, błąd ten jest zazwyczaj efektem zdefiniowania tej granicy (powód, dla którego się to robi, jest opisany w dalszej części rozdziału).

Są dwie główne przyczyny tego błędu. Pierwsza jest prosta — aplikacja wykorzystuje zbyt dużo klas, które nie mieszczą się w zdefiniowanej metaprzestrzeni (patrz rozdział 5., „Dobór wielkości metaprzestrzeni"). Druga przyczyna jest mniej oczywista: wyciek pamięci podczas ładowania klas. Najczęściej zdarza się to w serwerach dynamicznie ładujących klasy. Jednym z przykładów jest serwer aplikacji Java EE. Każda zainstalowana w nim aplikacja wykorzystuje własny obiekt ładujący klasy, dzięki czemu aplikacje są od siebie odizolowane, nie współdzielą klas i nie interferują ze sobą. W środowisku programistycznym, za każdym razem, gdy aplikacja jest modyfikowana, jest ponownie instalowana. Tworzony jest nowy obiekt ładujący klasy, a stary wychodzi poza zakres widoczności. Gdy tak się stanie, metadane klas mogą zostać usunięte. Jeżeli jednak obiekt ładujący klasy nie wyjdzie poza zakres widoczności, nie można usunąć metadanych klas. W efekcie metaprzestrzeń zapełnia się i pojawia się błąd. W takim przypadku można powiększyć metaprzestrzeń, choć w rzeczywistości spowoduje to tylko odłożenie problemu w czasie.

Jeżeli tego typu sytuacja ma miejsce w serwerze aplikacji, jedyne, co można zrobić, to poinformowanie dostawcy serwera o konieczności opracowania poprawki zapobiegającej wyciekowi pamięci. Jeżeli tworzona aplikacja wykorzystuje dużo obiektów ładujących klasy, należy upewnić się, czy je poprawnie usuwa, a w szczególności, czy kontekst żadnego wątku nie obejmuje tymczasowego obiektu ładującego klasy. Aby zdiagnozować tego typu przypadek, trzeba wykonać opisaną wcześniej analizę zrzutu sterty. W histogramie należy odszukać wszystkie instancje klasy `ClassLoader`, znaleźć ich obiekty główne i sprawdzić ich zawartość.

Kluczowe znaczenie w diagnostyce tego rodzaju przypadków ma — jak poprzednio — dokładna analiza komunikatu o przepełnieniu pamięci. Jeżeli metaprzestrzeń zostanie zapełniona, pojawi się następująca informacja:

```
Exception in thread "main" java.lang.OutOfMemoryError: Metaspace
```

Między innymi z powodu wycieku pamięci podczas ładowania klas należy określać maksymalną wielkość metaprzestrzeni. Jeżeli się tego nie zrobi, wtedy z powodu wycieku obiektów ładujących klasy zostanie zapełniona cała pamięć komputera.

Zapełnienie sterty

Gdy zapełni się sterta, pojawia się następujący komunikat:

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

Zazwyczaj przyczyna zapełnienia sterty jest taka sama, jak opisana wcześniej przyczyna zapełnienia metaprzestrzeni. Aplikacja po prostu potrzebuje większej sterty, ponieważ na aktualnie zdefiniowanej nie mieszczą się wszystkie wykorzystywane obiekty. Innym powodem może być wyciek pamięci, gdy aplikacja tworzy kolejne obiekty, a stare nie wychodzą poza zakres widoczności. W pierwszym przypadku powiększenie sterty może rozwiązać problem, natomiast w drugim tylko go odsunie

w czasie. W obu przypadkach jednak należy wykonać analizę zrzutu sterty w celu znalezienia obiektów, które zajmują najwięcej pamięci, natomiast rozwiązaniem jest zmniejszenie liczby lub wielkości tworzonych obiektów. Jeżeli w aplikacji ma miejsce wyciek pamięci, należy wykonać kilka zrzutów pamięci w kilkuminutowych odstępach i porównać je ze sobą. W tym celu mona wykorzystać wbudowaną funkcjonalność narzędzia Memory Analyzer. Po załadowaniu dwóch zrzutów można utworzyć histogramy i wyliczyć różnice między nimi.

Automatyczne zrzucanie pamięci

Błędy przepełnienia pamięci pojawiają się niespodziewanie, więc nigdy nie wiadomo, kiedy należy utworzyć zrzut sterty. Dlatego pojawiły się następujące flagi:

`-XX:+HeapDumpOnOutOfMemoryError`

Włączenie tej flagi (o domyślnej wartości `false`) powoduje, że maszyna JVM będzie tworzyła zrzut sterty za każdym razem, gdy pojawi się błąd przepełnienia pamięci.

`-XX:HeapDumpPath=<`*`ścieżka`*`>`

Za pomocą tej flagi określa się miejsce, w którym ma być utworzony zrzut. Domyślnie flaga zawiera nazwę bieżącego katalogu aplikacji. Ścieżka może być nazwą katalogu lub pliku. W pierwszym przypadku plik otrzymuje domyślną nazwę *java_pid<pid>.hprof*.

`-XX:+HeapDumpAfterFullGC`

Flaga powodująca utworzenie zrzutu po pełnym uporządkowaniu pamięci.

`-XX:+HeapDumpBeforeFullGC`

Flaga powodująca utworzenie zrzutu przed pełnym uporządkowaniem pamięci.

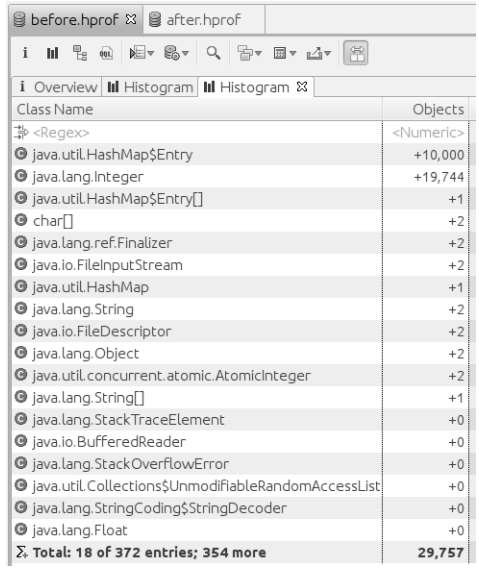
Jeżeli tworzonych jest kilka zrzutów, np. z powodu wielokrotnego pełnego porządkowania pamięci, nazwy plików są uzupełniane o kolejne numery.

Powyższych flag należy używać w sytuacjach, gdy aplikacja niespodziewanie wyświetla komunikaty o przepełnieniu pamięci i w celu ustalenia przyczyny problemu niezbędna jest analiza sterty. Należy pamiętać, że tworzenie zrzutu wprowadza pauzę w działaniu aplikacji, ponieważ zawartość sterty jest zapisywana na dysku.

Rysunek 7.5 przedstawia typowy przypadek wycieku pamięci spowodowany klasą kolekcji, tutaj `HashMap`. Tego rodzaju klasy przyczyniają się do przepełnienia pamięci najczęściej. Aplikacja umieszcza elementy w kolekcji, ale ich nie usuwa. Rysunek przedstawia histogram porównawczy, zawierający różnice w liczbach poszczególnych obiektów w dwóch zrzutach. Na przykład drugi zrzut zawiera 19 744 obiektów typu `Integer` więcej niż pierwszy.

Najskuteczniejszym sposobem rozwiązania problemu jest taka zmiana kodu aplikacji, aby elementy kolekcji były sukcesywnie usuwane, gdy przestaną być potrzebne. Ewentualnie można stosować kolekcje wykorzystujące słabe lub miękkie odwołania. Elementy takich kolekcji są automatycznie usuwane, gdy żadne obiekty się do nich nie odwołują. Kolekcje te mają jednak swoją cenę, o czym będzie mowa w dalszej części rozdziału.

Często zdarza się, że maszyna JVM po zgłoszeniu wyjątku kontynuuje działanie, ponieważ problem dotyczy tylko jednego wątku. Przeanalizujmy przykład aplikacji wykonującej obliczenia za pomocą dwóch wątków. W pewnym momencie jeden z nich zgłasza błąd `OutOfMemoryError`. Domyślnie procedura obsługi wątku wyświetla stos wywołań, a sam wątek kończy działanie.



Rysunek 7.5. Histogram porównawczy

Jednak drugi wątek pozostaje aktywny, więc maszyna JVM działa dalej. Ponieważ pierwszy wątek, w którym wystąpił błąd, zakończył działanie, prawdopodobnie przy następnym porządkowaniu pamięci duża jej ilość zostanie zwolniona. Będą to wszystkie obiekty wykorzystywane przez zakończony wątek, do których nie odwołuje się drugi, aktywny wątek. Zatem drugi wątek będzie działał dalej i będzie dysponował wystarczająco dużą stertą, aby dokończyć swoje zadania.

W opisany wyżej sposób funkcjonują platformy serwerowe wykorzystujące pule wątków do obsługi zapytań. Przechwytują błędy i zapobiegają dzięki temu przerywaniu działania wątków. Nie dotyczy to jednak opisywanego tu problemu, ponieważ pamięć, którą wątek wykorzystuje w celu obsłużenia zapytania, jest zwalniana podczas jej porządkowania.

Zatem błąd przepełnienia pamięci ma fatalne skutki jedynie wtedy, gdy dotyczy ostatniego wątku, innego niż wykorzystywany wewnętrznie przez maszynę JVM. W przypadku platformy serwerowej tak się jednak nie zdarza nigdy, a bardzo rzadko ma to miejsce w przypadku zwykłej aplikacji wykorzystującej kilka wątków. Zazwyczaj mechanizm obsługi błędów działa dobrze, a pamięć zajęta na potrzeby obsługi aktywnego zapytania jest często zwalniania podczas jej kolejnego porządkowania.

Jeżeli maszyna JVM ma kończyć działanie po przepełnieniu sterty, należy użyć flagi `-XX:+ExitOnOut` `↪OfMemoryError`, która domyślnie ma wartość `false`.

Zbyt długi czas porządkowania pamięci

Opisany w poprzednim podrozdziale mechanizm obsługi błędów opiera się na założeniu, że gdy zostanie zgłoszony błąd przepełnienia pamięci, cała wykorzystywana przez niego pamięć będzie zwolniona przy jej najbliższym porządkowaniu. Okazuje się, że założenie to nie zawsze jest słuszne.

Dochodzimy w ten sposób do ostatniej przyczyny błędu, tj. zbyt długiego czasu porządkowania pamięci, sygnalizowanego następującym komunikatem:

```
Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded
```

Błąd pojawia się wtedy, gdy są spełnione wszystkie poniższe warunki:

- Procentowy udział czasu poświęcanego na porządkowanie pamięci przekracza wartość określoną za pomocą flagi `-XX:GCTimeLimit=N`. Domyślnie flaga ta ma wartość 98, co oznacza, że pamięć jest porządkowana przez 98% czasu.
- Ilość pamięci zwalnianej podczas pełnego porządkowania pamięci jest mniejsza od wartości określonej za pomocą flagi `-XX:GCHeapFreeLimit=N`. Domyślnie flaga ta ma wartość 2 oznaczającą, że warunek jest spełniony, jeżeli zwalniane jest mniej niż 2% wielkości sterty.
- Powyższe warunki są spełnione po pięciu kolejnych porządkowaniach pamięci. Nie ma możliwości zmiany tej liczby.
- Flaga `-XX:+UseGCOverheadLimit` ma wartość `true` (domyślną).

Pamiętaj, że wszystkie powyższe warunki muszą być spełnione. Często zdarza się, że po pięciu pełnych porządkowaniach nie jest zgłaszany błąd przepełnienia pamięci, ponieważ mimo że aplikacja spędza ponad 98% czasu na porządkowaniu pamięci, za każdym razem odzyskuje więcej niż 2% wielkości sterty. W takim wypadku należy zwiększyć wartość flagi `GCHeapFreeLimit`.

Warto również wiedzieć, że jeżeli pierwsze dwa warunki będą spełnione po czterech kolejnych porządkowaniach, maszyna JVM w desperackiej próbie zwolnienia pamięci podczas piątego porządkowania usunie wszystkie miękkie odwołania do obiektów (o ile aplikacja je wykorzystuje). Zapobiega w ten sposób pojawieniu się błędu, ponieważ w ostatnim cyklu zwalnia więcej pamięci niż 2% wielkości sterty.

Krótkie podsumowanie

- Przyczyny błędu przepełnienia pamięci mogą być różne, dlatego nie można zakładać, że jest on oznaką problemu ze stertą.
- Najczęstszą przyczyną przepełnienia zarówno sterty, jak i metaprzestrzeni jest wyciek pamięci, który można zdiagnozować za pomocą narzędzi do analizy sterty.

Zmniejszenie zajętości pamięci

Pierwszym krokiem w ograniczeniu ilości wykorzystywanej pamięci jest zmniejszenie zajętości sterty. Stwierdzenie to nie powinno dziwić, bo mniejsze wykorzystanie sterty oznacza, że zapełnia się ona wolniej i rzadziej trzeba ją porządkować. Ponadto efekty mogą się kumulować. Jeżeli cykle porządkowania młodej generacji są rzadsze, obiekty dojrzewają wolniej, co z kolei oznacza mniejsze prawdopodobieństwo przeniesienia ich do starej generacji. Rzadziej są też wykonywane cykle pełnego porządkowania pamięci. Co więcej, mogą być wykonywane rzadziej, jeżeli podczas każdego z nich zwalniane jest więcej pamięci.

W tym podrozdziale opisane są trzy techniki ograniczania zajętości pamięci: zmniejszanie obiektów, ich leniwe inicjowanie i stosowanie kanonicznych wersji.

Zmniejszanie wielkości obiektów

Obiekty zajmują określoną część sterty, zatem najprostszym sposobem ograniczenia wykorzystania pamięci jest ich pomniejszenie. Zwiększenie sterty o 10% nie zawsze jest możliwe, ale ten sam efekt można osiągnąć pomniejszając połowę obiektów o 20%. Jak się dowiesz w rozdziale 12., wersja Java 11 optymalizuje obiekty typu `String`, dzięki czemu maksymalna wielkość sterty w porównaniu z wersją Java 8 jest nierzadko mniejsza o 25%. Optymalizacja nie wpływa na proces porządkowania pamięci ani na wydajności aplikacji.

Ilość zajmowanej przez obiekty pamięci można najprościej ograniczyć poprzez zmniejszenie ich liczby lub — co już nie jest takie proste — przez pomniejszenie samych obiektów. Tabela 7.1 przedstawia wielkości obiektów różnych typów.

Tabela 7.1. Wielkości (w bajtach) obiektów różnych typów

Typ	Wielkość
`byte`	1
`char`	2
`short`	2
`int`	4
`float`	4
`long`	8
`double`	8
Odwołanie	8 (lub 4 w 32-bitowej maszynie JVM dla systemu Windows)[a]

[a] Więcej szczegółowych informacji znajdziesz w podrozdziale „Skompresowane wskaźniki".

Termin „odwołanie" w ostatnim wierszu dotyczy dowolnego obiektu, np. tablicy lub instancji klasy. Podana liczba dotyczy pamięci zajmowanej tylko przez odwołanie. Wielkość obiektu zawierającego odwołania do innych obiektów zależy od tego, czy w analizie uwzględniane są płytkie, głębokie, czy zachowane obiekty. Ponadto obiekt zawiera niewidoczne pola nagłówków. W zwykłym obiekcie takie pole ma wielkość 8 bajtów w przypadku 32-bitowej maszyny na systemie Windows lub 16 bajtów w maszynie 64-bitowej, niezależnie od wielkości sterty. W tablicy pole nagłówka zajmuje 16 bajtów w maszynie 32-bitowej lub 64-bitowej ze stertą mniejszą niż 32 GB, lub 24 bajty w pozostałych przypadkach.

Przeanalizujmy poniższe definicje klas:

```
public class A {
    private int i;
}

public class B {
    private int i;
    private Locale l = Locale.US;
}

public class C {
    private int i;
    private ConcurrentHashMap chm = new ConcurrentHashMap();
}
```

Tabela 7.2 przedstawia wielkości pojedynczych instancji tych klas w przypadku 64-bitowej maszyny JVM ze stertą mniejszą niż 32 GB.

Tabela 7.2. Wielkości (w bajtach) prostych obiektów

	Płytki	Głęboki	Zachowany
A	16	16	16
B	24	216	24
C	24	200	200

Odwołanie typu `Locale` w klasie `B` powiększa obiekt o 8 bajtów. Jednak w tym przykładzie obiekt typu `Locale` jest współdzielony przez inne obiekty. Jeżeli obiekt ten nie jest nigdzie wykorzystywany, oznacza to, że wraz z odwołaniem niepotrzebnie zajmuje pamięć. Jeżeli aplikacja tworzy wiele obiektów typu `B`, niewykorzystane bajty sumują się.

Następnym przykładem jest odwołanie typu `ConcurrentHashMap`, zajmujące określoną liczbę bajtów wraz z odwołaniami do innych obiektów. Jeżeli instancje klasy `C` nie są wykorzystywane, niepotrzebnie zajmują mnóstwo miejsca w pamięci.

Jednym ze sposobów pomniejszania obiektów jest definiowanie tylko niezbędnych klas. Mniej oczywisty sposób polega na korzystaniu z mniejszych typów danych. Jeżeli w klasie trzeba np. przechowywać informacje o jednym z ośmiu możliwych stanów, należy użyć typu `byte`, a nie `int`, dzięki czemu zaoszczędzi się 3 bajty. Stosując typ `float` zamiast `double` lub `int` zamiast `long` można zaoszczędzić sporo pamięci, szczególnie jeżeli tworzonych jest wiele instancji danej klasy. Jak się dowiesz z rozdziału 12., podobne oszczędności uzyskuje się stosując kolekcje o odpowiedniej wielkości lub zwykłe instancje zamiast kolekcji.

Wyrównanie i wielkość obiektów

Wszystkie klasy przedstawione w tabeli 7.2 zawierają pole typu `int`, o którym do tej pory nie wspominałem. Po co ono jest?

Tylko po to, aby uprościć opis poszczególnych klas, np. że klasa `B` zajmuje 8 bajtów więcej niż `A` (jest to zgodne z oczekiwaniami, a konkluzja jest bardziej zrozumiała).

Pojawia się tu pewien ważny szczegół: każdy obiekt jest powiększany o dodatkowe bajty, aby jego wielkość była wielokrotnością liczby 8. Gdyby klasa `A` nie zawierała pola `i`, jej instancja i tak zajmowałaby 16 bajtów. Dodatkowe 4 bajty stanowiłyby uzupełnienie obiektu, aby jego wielkość była wielokrotnością liczby 8. Gdyby klasa `B` nie zawierała pola `i`, jej instancja również zajmowałaby 16 bajtów, tj. tyle samo, co `A`. Uzupełnienie to powoduje, że instancja klasy `B` jest o 8 bajtów większa od instancji klasy `A`, mimo że `B` zawiera tylko jedno dodatkowe, 4-bajtowe odwołanie.

Maszyna JVM uzupełnia w ten sposób również obiekty zajmujące nieparzystą liczbę bajtów, dzięki czemu tablice złożone z takich obiektów są dopasowane do granic segmentów pamięci wykorzystywanego komputera.

Podsumowując, usunięcie niektórych pól lub zmniejszenie obiektu może, ale nie musi przysparzać oszczędności, ale nie ma powodu, aby tego nie robić.

Ze strony projektu OpenJDK (*https://oreil.ly/cSPvd*) można pobrać narzędzie do obliczania wielkości obiektów.

Usuwając pola z obiektu można go pomniejszyć, ale wciąż pozostaje dylemat: czy należy stosować pola zawierające przetworzone dane? Jest to typowy w programowaniu kompromis czas-przestrzeń: lepiej poświęcić więcej pamięci (przestrzeni), aby przechowywać przetworzone dane, czy więcej cykli procesora (czasu) na przetwarzanie danych za każdym razem, gdy będzie zachodziła taka potrzeba? W przypadku Javy przestrzeń jest związana z czasem, ponieważ porządkowanie dodatkowej pamięci zajmuje więcej cykli procesora.

Na przykład kod mieszający obiektu typu `String` wylicza się sumując w określony sposób kody wszystkich znaków, co trwa dłuższą chwilę. Dlatego wartość tę wylicza się tylko raz i zapisuje w zmiennej instancji. Dzięki temu niemal zawsze, gdy wykorzystywana jest ta wartość, wzrost wydajności aplikacji jest na tyle duży, że warto jest poświęcić dodatkową pamięć. Z drugiej strony metoda `toString()` w większości klas nie zapisuje tekstowej reprezentacji obiektu w zmiennej instancji, ponieważ zajmowana byłaby dodatkowa pamięć zarówno przez zmienną, jak i ciąg znaków. W takim przypadku zazwyczaj większą wydajność uzyskuje się generując za każdym razem ciąg znaków od nowa niż zapisując ciąg w pamięci. Należy też pamiętać, że kod mieszający obiektu typu `String` jest wykorzystywany często, a wynik metody `toString()` dość rzadko.

Wszystko oczywiście zależy od sytuacji, a decyzja o rezygnacji z zapisywania wyników w pamięci na rzecz ich wielokrotnego wyliczania lub odwrotnie jest bardzo płynna i uzależniona od wielu czynników. Jeżeli celem jest skrócenie czasu porządkowania pamięci, bardziej korzystne okazuje się wyliczanie wyników.

Krótkie podsumowanie

- Często wydajność porządkowania pamięci można poprawić zmniejszając wielkości obiektów.
- Wielkość obiektów nie zawsze jest oczywista, ponieważ są one uzupełniane o dodatkowe bajty tak, aby były dopasowane do 8-bajtowych segmentów pamięci. Ponadto odwołania do obiektów w przypadku 32-bitowej maszyny JVM mają inną wielkość niż w maszynie 64-bitowej.
- Pamięć zajmuje nawet pusta zmienna instancji.

Leniwe inicjowanie obiektu

Zazwyczaj decyzja o utworzeniu zmiennej instancji nie jest tak jednoznaczna, jak sugeruje poprzedni podrozdział. Jeżeli na przykład klasa wykorzystuje obiekt typu `Calendar` tylko przez 10% czasu, powinna go zapisać w pamięci, a nie tworzyć za każdym razem na nowo, ponieważ jest to bardzo czasochłonna operacja. W takich sytuacjach można wykorzystać **leniwe inicjowanie** obiektów (ang. *lazy initialization*).

We wszystkich dotychczasowych opisach przyjąłem założenie, że zmienne instancji są inicjowanie aktywnie. Klasa wykorzystująca obiekt typu `Calendar`, która nie musi być wątkowo bezpieczna, może wyglądać następująco:

```
public class CalDateInitialization {
  private Calendar calendar = Calendar.getInstance();
  private DateFormat df = DateFormat.getDateInstance();
```

```
  private void report(Writer w) {
    w.write("Godzina " + df.format(calendar.getTime()) + ": " + this);
  }
}
```

Leniwe inicjowanie obiektu pogarsza nieznacznie wydajność aplikacji, ponieważ przed każdym użyciem obiektu trzeba sprawdzać jego stan. Poniżej przedstawiony jest odpowiednio zmieniony kod klasy:

```
public class CalDateInitialization {
  private Calendar calendar;
  private DateFormat df;

  private void report(Writer w) {
    if (calendar == null) {
      calendar = Calendar.getInstance();
      df = DateFormat.getDateInstance();
    }
    w.write("Godzina " + df.format(calendar.getTime()) + ": " + this);
  }
}
```

Leniwe inicjowanie obiektu przynosi najwięcej korzyści wtedy, gdy operacje na obiekcie są wykonywane rzadko. W przeciwnym wypadku oszczędność pamięci będzie żadna, ponieważ obiekt będzie nieustannie alokowany, co dodatkowo, choć nieznacznie, pogorszy wydajność aplikacji.

Jeżeli kod musi być bezpieczny wątkowo, leniwe inicjowanie zaczyna się komplikować. W pierwszym kroku najprościej jest zastosować tradycyjne synchronizowanie wątków:

```
public class CalDateInitialization {
  private Calendar calendar;
  private DateFormat df;

  private synchronized void report(Writer w) {
    if (calendar == null) {
      calendar = Calendar.getInstance();
      df = DateFormat.getDateInstance();
    }
    w.write("Godzina " + df.format(calendar.getTime()) + ": " + this);
  }
}
```

Wprowadzenie synchronizacji może pogorszyć wydajność aplikacji, lecz jest to mało prawdopodobne. Leniwe inicjowanie obiektów przynosi korzyści tylko wtedy, gdy obiekty są wykorzystywane rzadko. Jeżeli jednak są zawsze inicjowane, nie ma żadnych oszczędności pamięci. Zatem synchronizacja wraz z leniwym inicjowaniem obiektów pogarszają wydajność aplikacji, jeżeli obiekty te są wykorzystywane przez wiele wątków jednocześnie. Takie sytuacje zdarzają się rzadko, ale nie można ich wykluczać.

Problemu z synchronizacją nie ma jedynie wtedy, gdy leniwie inicjowane obiekty są bezpieczne wątkowo. Klasa `DateFormat` nie spełnia tego warunku, więc w poniższym przykładzie nie ma znaczenia, czy blokada obejmuje obiekt typu `Calendar`, czy nie. Jeżeli leniwie inicjowany obiekt jest intensywnie wykorzystywany, niezbędna synchronizacja dostępu do obiektu typu `DateFormat` i tak jest problemem. Kod bezpieczny wątkowo może wyglądać następująco:

Leniwe inicjowanie obiektów a wydajność aplikacji

Nie zawsze leniwe inicjowanie obiektów pogarsza wydajność aplikacji. Przeanalizujmy przykład zawartej w pakiecie JDK klasy `ArrayList`, która obsługuje zawartą w niej tablicę obiektów. W starszych wersjach Javy pseudokod tej klasy wyglądał następująco:

```
public class ArrayList {
  private Object[] elementData = new Object[16];
  int index = 0;

  public void add(Object o) {
    ensureCapacity();
    elementData[index++] = o;
  }

  private void ensureCapacity() {
    if (index == elementData.length) {
      ...Ponowna alokacja tablicy i kopiowanie danych...
    }
  }
}
```

Kilka lat temu klasa ta została zmieniona i teraz tablica `elementData` jest leniwie inicjowana. Ponieważ metoda `ensureCapacity()` sprawdza wcześniej wielkość tablicy, wydajność innych metod nie pogorszyła się. Podczas inicjowania i powiększania tablicy wykorzystywany jest ten sam kod. Zmieniona klasa wykorzystuje statyczną tablicę o zerowej wielkości, dzięki czemu jej wydajność nie zmieniła się:

```
public class ArrayList {
  private static final Object[] EMPTY_ELEMENTDATA = {} ;
  private Object[] elementData = EMPTY_ELEMENTDATA;
}
```

Oznacza to, że metodę `ensureCapacity()` można w zasadzie pozostawić bez zmian, ponieważ na początku zarówno zmienna `index`, jak i `elementData.length` zawierają wartość 0.

```
public class CalDateInitialization {
  private Calendar calendar;
  private DateFormat df;

  private void report(Writer w) {
    unsychronizedCalendarInit();
    synchronized(df) {
      w.write("Goczina " + df.format(calendar.getTime()) + ": " + this);
    }
  }
}
```

Jeżeli leniwie inicjowany obiekt nie jest bezpieczny wątkowo, można synchronizować dostęp do zmiennej np. stosując opisaną wcześniej metodę z modyfikatorem `synchronized`.

Przeanalizujmy nieco inny przykład, w którym leniwie inicjowany jest duży obiekt typu `Concurrent` ↪`HashMap`:

```
public class CHMInitialization {
  private ConcurrentHashMap chm;

  public void doOperation() {
```

```
      synchronized(this) {
        if (chm == null) {
          chm = new ConcurrentHashMap();
          ... Kod wypełniający mapę ...
        }
      }
      ... Kod wykorzystujący zmienną chm ...
    }
  }
```

Do obiektu typu `ConcurrentHashMap` może się odwoływać kilka wątków, więc jest to jeden z niewielu przypadków poprawnego leniwego inicjowania obiektu, w którym wprowadzenie dodatkowej synchronizacji może pogorszyć wydajność aplikacji. Jest to jednak rzadki przypadek i jeżeli obiekt nie jest często wykorzystywany, warto się zastanowić, czy w ogóle trzeba go leniwie inicjować. Problem można rozwiązać dwukrotnie sprawdzając blokadę:

```
public class CHMInitialization {
  private volatile ConcurrentHashMap instanceChm;
  public void doOperation() {
    ConcurrentHashMap chm = instanceChm;
    if (chm == null) {
      synchronized(this) {
        chm = instanceChm;
        if (chm == null) {
          chm = new ConcurrentHashMap();
          ... Kod wypełniający mapę ...
          instanceChm = chm;
        }
      }
      ... Kod wykorzystujący zmienną chm ...
    }
  }
}
```

Pojawia się tu ważny problem związany z wątkami. Zmienna instancji musi być zadeklarowana jako `volatile`. Wydajność można nieznacznie poprawić przypisując zmienną instancji lokalnej zmiennej. Więcej szczegółowych informacji na ten temat będzie podanych w rozdziale 9. W rzadkich przypadkach, w których uzasadnione jest leniwe inicjowanie obiektów w wielowątkowym kodzie, należy stosować opisaną technikę.

Aktywne zamykanie obiektów

Przeciwieństwem leniwego inicjowania obiektu jest jego **aktywne zamykanie** (ang. *eager deinitialization*) poprzez przypisanie zmiennej wartości `null`. Dzięki temu obiekt może być szybciej usunięty przez kolektor. W teorii brzmi to nieźle, ale w praktyce sprawdza się w nielicznych, szczególnych przypadkach.

Mogłoby się wydawać, że obiekt, który można leniwie inicjować, można również aktywnie zamykać. Użytym w poprzednim przykładzie zmiennym typów `Calendar` i `DateFormat` można przypisać wartość `null` na końcu metody `report()`. Jeżeli jednak zmienne te nie będą wykorzystywane w kolejnych wywołaniach metody lub w innych miejscach klasy, nie ma powodu, aby w ogóle były to zmienne instancji. Zamiast nich można wewnątrz metody użyć lokalnych zmiennych, które po zakończeniu wykonywania kodu wyjdą poza zakres widoczności i zostaną usunięte przez kolektor.

Najczęściej wyjątki od zasady unikania aktywnego zamykania obiektów zdarzają się w klasach, które przez długi czas utrzymują odwołania do obiektów, po czym otrzymują sygnał, że obiekty te nie są już potrzebne. Przeanalizujmy przedstawioną niżej implementację metody `remove()` w klasie `ArrayList` zawartej w pakiecie JDK (część kodu została uproszczona):

```
public E remove(int index) {
   E oldValue = elementData(index);
  int numMoved = size - index - 1;
  if (numMoved > 0)
    System.arraycopy(elementData, index+1,
                     elementData, index, numMoved);
  elementData[--size] = null; // Zamknięcie obiektu, aby kolektor mógł go usunąć.
  return oldValue;
}
```

W kodzie źródłowym pakietu JDK, generalnie opatrzonego bardzo nielicznymi komentarzami, znajduje się komentarz do wiersza, w którym zmiennej jest przypisywana wartość `null`. Jest to tak nietypowy przypadek, że zasługuje na specjalny opis. Prześledźmy, co się dzieje, gdy usuwany jest ostatni element tablicy. Zmniejsza się liczba jej elementów zapisana w zmiennej instancji `size`. Załóżmy, że zmienna ta zmniejszyła swoją wartość z 5 do 4. Po tej operacji nie można się odwołać do elementu `elementData[4]`, ponieważ znajduje się on poza granicą tablicy. Zatem odwołanie `elementData[4]` przestaje być nieaktualne. Sama tablica `elementData` będzie prawdopodobnie aktywna jeszcze przez jakiś czas, więc odwołania do wszystkich obiektów, które nie są potrzebne, należy aktywnie ustawić na `null`.

Kluczowe znaczenie ma pojęcie nieaktualnego odwołania. Jeżeli tworzona klasa będzie przez dłuższy czas tworzyła i usuwała obiekty, należy zwrócić szczególną uwagę, aby nie pojawiały się w niej nieaktualne odwołania. Jawne przypisanie odwołaniu wartości `null` pozwoli również nieznacznie poprawić wydajność aplikacji.

Krótkie podsumowanie

- Leniwe inicjowanie obiektów należy stosować tylko wtedy, gdy często wykorzystywany kod nie odwołuje się do tych obiektów.
- W bezpiecznym wątkowo kodzie rzadko pojawia się potrzeba leniwego inicjowania obiektów i zazwyczaj można wykorzystać istniejącą synchronizację.
- W kodzie bezpiecznym wątkowo należy dwukrotnie sprawdzać blokadę leniwie inicjowanego obiektu.

Obiekty niemutowalne i kanoniczne

W Javie wiele typów jest niemutowalnych. Dotyczy to typów prymitywnych, takich jak `Integer`, `Double` czy `Boolean`, jak również bardziej złożonych, np. `BigDecimal`. Najczęściej używanym, niemutowalnym typem jest oczywiście `String`. Z programistycznego punktu widzenia dobrą praktyką jest definiowanie własnych, niemutowalnych klas.

Jeżeli obiekty wyżej wymienionych typów są często tworzone i natychmiast usuwane, ich wpływ na porządkowanie obszaru młodej generacji jest niewielki, o czym się przekonałeś w rozdziale 5. Jednak jeżeli obiekty niemutowalne są przenoszone do starej generacji, wydajność aplikacji może się pogorszyć.

Nie ma powodów, aby unikać stosowania typów niemutowalnych, nawet jeżeli wydaje się to nielogiczne, że obiektów tych nie można zmieniać i trzeba je wciąż tworzyć na nowo. Jedną z technik optymalizacji stosowania tego rodzaju obiektów jest zapobieganie tworzeniu wielokrotnych kopii tego samego obiektu.

Najlepszym przykładem jest klasa `Boolean`. Każda aplikacja potrzebuje tylko dwóch instancji tej klasy, reprezentujących odpowiednio wartości `true` i `false`. Niestety, klasa ta została źle zaprojektowana. Ponieważ posiada publiczny konstruktor, można tworzyć dowolną liczbę jej instancji, mimo że każda z nich może być tylko jednym z dwóch kanonicznych obiektów. Lepiej byłoby, gdyby klasa ta posiadała prywatny konstruktor i statyczne metody zwracające w zależności od parametru wartości `Boolean.TRUE` lub `Boolean.FALSE` (a jeszcze lepiej, gdyby w ogóle nie trzeba było tworzyć obiektów typu `Boolean`). Własne niemutowalne klasy funkcjonujące w ten sposób nie zajmowałyby sterty.

Tego rodzaju niemutowalne obiekty reprezentujące skończoną liczbę wartości są nazywane **obiektami kanonicznymi**.

Tworzenie obiektów kanonicznych

Pamięć można oszczędzić stosując obiekty kanoniczne. Pakiet JDK oferuje specjalnie przeznaczoną do tego celu funkcjonalność. Za pomocą metody `intern()` można znaleźć kanoniczną wersję ciągu znaków. Więcej szczegółowych informacji na ten temat znajdziesz w rozdziale 12., natomiast teraz dowiesz się, jak można ten efekt osiągnąć w przypadku własnych klas.

Aby utworzyć obiekt kanoniczny, należy zdefiniować mapę zawierającą jego kanoniczne wersje. Aby zapobiec wyciekowi pamięci, odwołania do tych obiektów muszą być słabe. Poniżej przedstawiony jest szkielet takiej klasy:

```
public class ImmutableObject {
  private static WeakHashMap<ImmutableObject, ImmutableObject>
    map = new WeakHashMap();
  public ImmutableObject canonicalVersion(ImmutableObject io) {
    synchronized(map) {
      ImmutableObject canonicalVersion = map.get(io);
      if (canonicalVersion == null) {
        map.put(io, new WeakReference(io));
        canonicalVersion = io;
      }
      return canonicalVersion;
    }
  }
}
```

W aplikacji wielowątkowej może pojawić się problem z synchronizacją dostępu, którego nie można rozwiązać w prosty sposób, jeżeli wykorzystywane są klasy zawarte w pakiecie JDK. Pakiet ten nie oferuje mapy ze słabymi odwołaniami, do której kilka wątków mogłoby się odwoływać jednocześnie. Pojawiły się jednak sugestie, m.in. w żądaniu JSR (ang. *Java Specification Request* — żądanie zmiany specyfikacji Javy) nr 166, aby rozszerzyć pakiet JDK o klasę `CustomConcurrentHashMap`. W internecie można znaleźć kilka niezależnych implementacji tej klasy.

Krótkie podsumowanie

- Obiekty niemutowalne można kanonizować i zarządzać w specjalny sposób ich cyklem życia.
- Kanonizując obiekty można wyeliminować ich wielokrotne niemutowalne kopie i w ten sposób znacznie ograniczyć wykorzystanie sterty.

Zarządzanie cyklem życia obiektów

Drugim obszernym tematem związanym z wykorzystaniem pamięci jest *zarządzanie cyklem życia obiektów*. Java w dużej mierze zwalnia programistę z tego obowiązku. Gdy utworzone w aplikacji obiekty przestają być potrzebne, wychodzą poza zakres widoczności i kolektor usuwa je z pamięci.

Czasami jednak taki cykl życia obiektów nie jest optymalny. Tworzenie obiektów niektórych typów jest czasochłonne i samodzielnie zarządzając ich cyklami życia można zwiększyć wydajność aplikacji, nawet jeżeli kolektor będzie musiał wykonywać dodatkowe operacje. W tym rozdziale opisuję, kiedy należy zmieniać cykl życia obiektów poprzez ich wielokrotne wykorzystywanie lub odwoływanie się do nich w specjalny sposób.

Obiekty wielokrotnego użytku

Obiekty można wykorzystywać wielokrotnie na dwa sposoby: tworząc pule obiektów lub lokalne zmienne wątkowe. Inżynierowie zajmujący się porządkowaniem pamięci mogą w tym momencie zaprotestować, ponieważ obie techniki obniżają skuteczność tego procesu. Szczególnie niepopularna jest pierwsza technika, która z kilku innych powodów nie jest też lubiana przez innych programistów, niezwiązanych z porządkowaniem pamięci.

Jeden z powodów tej niechęci wydaje się oczywisty: obiekty wielokrotnego użytku przez długi czas zajmują stertę. Jeżeli jest ich dużo, mniej miejsca pozostaje dla nowych obiektów i częściej trzeba porządkować stertę. Ale to dopiero początek problemów.

Jak pamiętasz z rozdziału 6., obiekt jest tworzony w edenie. Podczas kilku pierwszych cykli porządkowania obszaru młodej generacji obiekt jest przenoszony z jednego do drugiego obszaru obiektów ocalonych, po czym ostatecznie ląduje w obszarze starej generacji. Za każdym razem, gdy na nowym lub ostatnio utworzonym obiekcie w puli są wykonywane jakieś operacje, obiekt ten jest kopiowany i aktualizowane są odwołania do niego, do chwili ostatecznego umieszczenia go w starej generacji.

Przeniesienie obiektu do starej generacji pozornie oznacza koniec zamieszania, ale w rzeczywistości pojawiają się wtedy problemy z wydajnością aplikacji. Czas potrzebny na pełne uporządkowanie pamięci jest proporcjonalny do liczby aktywnych obiektów starej generacji. Większe znacznie od wielkości sterty ma ilość danych. Krócej trwa porządkowanie obszaru starej generacji o wielkości 3 GB zajmowanego przez kilka aktywnych obiektów niż obszaru o wielkości 1 GB, w którym aktywnych jest 75% wszystkich obiektów.

Stosowanie kolektora współbieżnego i zapobieganie pełnemu porządkowaniu pamięci nie poprawia sytuacji, ponieważ czas potrzebny na oznakowanie obiektów jest podobnie proporcjonalny do ilości aktywnych danych. Jeżeli wykorzystywany jest kolektor CMS, obiekty znajdujące się w puli mogą

Skuteczność procesu porządkowania pamięci

Jaki dokładnie wpływ ma wielkość aktywnych obiektów na czas porządkowania pamięci? Najprostsza odpowiedź brzmi: ogromny.

Poniżej przedstawiony jest fragment dziennika utworzonego przez kolektor podczas testu aplikacji. Test był wykonany za pomocą typowego komputera wyposażonego w system Linux i czterordzeniowy procesor. Sterta miała wielkość 4 GB, z czego obszar młodej generacji miał stałą wielkość równą 1 GB.

```
[Full GC [PSYoungGen: 786432K->786431K(917504K)]
        [ParOldGen: 3145727K->3145727K(3145728K)]
        3932159K->3932159K(4063232K)
        [PSPermGen: 2349K->2349K(21248K)], 0.5432730 secs]
        [Times: user=1.72 sys=0.01, real=0.54 secs]
...
[Full GC [PSYoungGen: 786432K->0K(917504K)]
        [ParOldGen: 3145727K->210K(3145728K)]
        3932159K->210K(4063232K)
        [PSPermGen: 2349K->2349K(21248K)], 0.0687770 secs]
        [Times: user=0.08 sys=0.00, real=0.07 secs]
...
[Full GC [PSYoungGen: 349567K->349567K(699072K)]
        [ParOldGen: 3145727K->3145727K(3145728K)]
        3495295K->3495295K(3844800K)
        [PSPermGen: 2349K->2349K(21248K)], 0.7228880 secs]
        [Times: user=2.41 sys=0.01, real=0.73 secs]
```

Zwróć uwagę na środkowy wpis. Podczas porządkowania sterty została usunięta większość odwołań do obiektów starej generacji, dzięki czemu zajmowany przez nią obszar zmniejszył się do zaledwie 210 kB. Operacja trwała tylko 70 ms. W pozostałych przypadkach większość obiektów na stercie była wciąż w użyciu, dlatego cykle pełnego porządkowania pamięci, podczas których usuwanych było niewiele danych, trwały od 540 ms do 730 ms. Na szczęście, w tym teście pamięć porządkowały cztery wątki, ale w jednordzeniowym systemie krótkie porządkowanie trwało 80 ms, natomiast długie 2410 ms, tj. ponad 30 razy dłużej.

być przenoszone do starej generacji w różnych momentach, co skutkuje fragmentacją puli i zwiększa prawdopodobieństwo awarii trybu współbieżnego. Podsumowując, im dłużej obiekty znajdują się na stercie, tym mniej wydajne jest jej porządkowanie.

Zatem obiekty wielokrotnego użytku są czymś złym. Możemy się więc teraz zastanowić, kiedy i jak je należy stosować.

Pakiet JDK oferuje kilka podstawowych pul obiektów: pulę wątków, którą opiszę w rozdziale 9., oraz miękkie odwołania. **Miękkie odwołania**, opisane w dalszej części rozdziału, są w rzeczywistości ogromną pulą obiektów wielokrotnego użytku. Aplikacja serwerowa wykorzystuje pulę obiektów reprezentujących połączenia z bazą danych i innym zasobami. Podobnie sytuacja wygląda w przypadku lokalnych zmiennych wątkowych. Pakiet JDK zawiera mnóstwo klas, które wykorzystują tego rodzaju zmienne, aby uniknąć wielokrotnego alokowania obiektów określonych typów. Zatem nawet eksperci od Javy zgadzają się z koniecznością stosowania w pewnych sytuacjach obiektów wielokrotnego użytku.

Tworzenie niektórych obiektów jest tak czasochłonne, że wielokrotne ich używając można poprawić wydajność aplikacji pomimo wydłużenia czasu porządkowania pamięci. Dotyczy to w m.in. puli połączeń JDBC (ang. *Java Database Connectivity* — łącze do baz danych w języku Java). Nawiązanie

połączenia sieciowego, zalogowanie do bazy danych i utworzenie sesji trwa jakiś czas. W takim wypadku stosując pulę obiektów można radykalnie zwiększyć wydajność aplikacji. Wątki umieszcza się w puli, dzięki czemu nie traci się czasu na ich tworzenie. Generator liczb losowych wykorzystuje lokalne zmienne wątkowe, aby skrócić czas inicjacji itd.

Wspólną cechą opisanych przykładów jest długi czas inicjowania obiektu. Obiekt w Javie *tworzony* jest szybko i ten argument jest często podnoszony przez przeciwników wielokrotnego wykorzystywania obiektów. Czas *inicjowania* obiektu jest jego indywidualną cechą. Dlatego obiekty wielokrotnego użytku należy tworzyć wtedy, gdy ich inicjowanie jest czasochłonne i ma dominujący wpływ na działanie aplikacji.

Inną wspólną cechą powyższych przykładów jest zależność wydajności procesu porządkowania pamięci od liczby obiektów wielokrotnego użytku. Jeżeli jest ich dużo, wtedy porządkowanie pamięci znacząco się wydłuża.

Poniżej wymienionych jest kilka przykładów obiektów wielokrotnego użytku wraz z opisem, gdzie i kiedy są one stosowane w pakiecie JDK i aplikacjach:

Pule wątków

Inicjowanie wątku jest czasochłonne.

Pule połączeń JDBC

Nawiązanie połączenia z bazą danych jest bardzo czasochłonne.

Duże tablice

Gdy tworzona jest tablica, wszystkie jej elementy są inicjowane domyślną wartością zależną od typu tablicy, np. `null`, 0 lub `false`. Jeżeli tablica jest duża, może to być czasochłonny proces.

Natywne bufory NIO

Jawne tworzenie bufora `java.nio.Buffer` poprzez wywołanie metody `allocateDirect()` jest czasochłonną operacją, niezależnie od wielkości bufora. Dlatego dobrą praktyką jest tworzenie jednego dużego bufora i dzielenie go na części wielokrotnego użytku.

Klasy szyfrujące

Inicjowanie obiektów związanych z bezpieczeństwem danych, np. typu `MessageDigest` lub `Signature`, jest czasochłonne.

Obiekty kodujące i dekodujące ciągi znaków

Wiele klas zawartych w pakiecie JDK wykorzystuje tego rodzaju obiekty. W większości są to słabe odwołania, o czym będzie mowa w dalszej części rozdziału.

Pomocnicze obiekty `StringBuilder`

Klasa `BigDecimal` wykorzystuje obiekt typu `StringBuilder` do wyliczania pośrednich wyników.

Generatory liczb losowych

Inicjowanie obiektów typu `Random`, a szczególnie `SecureRandom`, jest bardzo czasochłonne.

Zapytania DNS

Zapytania sieciowe są długotrwałe.

Kodery i dekodery ZIP

Inicjowanie powyższych obiektów, co ciekawe, nie jest szczególnie czasochłonne. Dużo czasu zajmuje jednak ich usuwanie, ponieważ obiekty te stosują finalizację, aby zwolnić natywną pamięć. Więcej szczegółowych informacji na ten temat znajduje się w podrozdziale „Finalizery i odwołania finalne".

Pule obiektów i lokalne zmienne wątkowe różnie wpływają na wydajność aplikacji. Przyjrzyjmy się im teraz dokładniej.

Pule obiektów

Pule obiektów nie są lubiane przez programistów z kilku powodów, z których tylko nieliczne dotyczą wydajności. Przede wszystkim trudno jest określić właściwą wielkość puli, a dodatkowo zarządzanie nią spada na programistę. Zanim obiekt wyjdzie poza zakres widoczności, trzeba pamiętać o umieszczeniu go z powrotem w puli.

Skupmy się jednak na wpływie puli obiektów na wydajność aplikacji, a dokładniej na wymienionych niżej aspektach.

Porządkowanie pamięci

Jak już się przekonałeś, duża liczba obiektów może drastycznie pogorszyć wydajność procesu porządkowania pamięci.

Synchronizacja dostępu

Dostęp do puli obiektów musi być zsynchronizowany. Jeżeli obiekty w puli są często zmieniane lub z niej usuwane, mogą pojawić się spiętrzenia w dostępie do nich. W efekcie uzyskanie dostępu do obiektu może zajmować więcej czasu niż jego utworzenie.

Ograniczanie dostępu

Pula obiektów może mieć korzystny wpływ na wydajność aplikacji, ponieważ można ją wykorzystywać do ograniczania dostępu do deficytowych zasobów. Jak się dowiedziałeś w rozdziale 2., obciążenie systemu ponad graniczną wartość obniża jego wydajność. Jest to jeden z powodów, dla którego warto stosować pulę obiektów. Jeżeli jednocześnie zostanie uruchomionych zbyt wiele wątków, procesor zostanie przeciążony i wydajność aplikacji spadnie. Ten przypadek będzie opisany w rozdziale 9.

Opisywana sytuacja dotyczy również dostępu do zewnętrznych systemów, w szczególności połączeń JDBC. Jeżeli zostanie nawiązanych więcej połączeń, niż baza będzie w stanie obsłużyć, jej wydajność pogorszy się. W takich sytuacjach warto ograniczać dostęp do zasobów (np. połączeń JDBC) zmniejszając pulę. Wydajność całego systemu może się poprawić mimo tego, że wątki aplikacji będą musiały czekać na uzyskanie dostępu.

Lokalne zmienne wątkowe

Lokalne zmienne wątkowe mają wpływ na następujące aspekty wydajności aplikacji:

Zarządzanie cyklem życia obiektów

Zarządzanie lokalnymi zmiennymi wątkowymi jest znacznie prostsze i mniej czasochłonne niż zarządzanie pulami obiektów. W obu przypadkach trzeba uzyskać obiekt, tj. pobrać go z puli lub wywołać metodę `get()` lokalnej zmiennej. Jednak w przypadku puli trzeba zwalniać blokadę obiektu, gdy przestaje on być potrzebny. W przeciwnym wypadku inny wątek nie będzie mógł go użyć. Zmienne wątkowe są zawsze dostępne i nie trzeba ich jawnie odblokowywać.

Kardynalność

Zazwyczaj liczba lokalnych zmiennych wątkowych jest równa liczbie wątków. Są jednak wyjątki od tej zasady. Zmienna lokalna jest tworzona dopiero przed jej pierwszym użyciem. Może się więc zdarzyć, że zmiennych będzie mniej niż wątków (ale nie więcej). Przez większość czasu liczby te są równe.

Natomiast pula obiektów może mieć dowolną wielkość. Jeżeli do obsługi zapytania czasami jest potrzebne jedno połączenie JDBC, a czasami dwa, wtedy dobiera się odpowiednią pulę, np. dwunastu połączeń dla ośmiu wątków. W przypadku lokalnych zmiennych wątkowych nie jest to możliwe, podobnie jak ograniczanie za ich pomocą dostępu do zasobów (chyba że zmniejszy się liczbę wątków).

Synchronizacja

Dostępu do lokalnych zmiennych wątkowych nie trzeba synchronizować, ponieważ z każdej zmiennej może korzystać tylko jeden wątek, a metoda `get()` jest dość szybka. Jednak nie zawsze tak było. W pierwszych wersjach Javy uzyskanie dostępu do zmiennej było czasochłonne. Jeśli w przeszłości unikałeś korzystania ze zmiennych lokalnych z powodu ich negatywnego wpływu na wydajność aplikacji, rozważ ich użycie w nowych wersjach Javy.

Synchronizacja dostępu jest ciekawym zagadnieniem, ponieważ wzrost wydajności aplikacji wynika głównie z braku konieczności synchronizowania dostępu do zmiennych lokalnych (co jest niezbędne w przypadku obiektów wielokrotnego użytku). Przeanalizujmy przykład użycia klasy `Thread` ↪`LocalRandom` w przykładowej aplikacji przetwarzającej dane giełdowe. Klasa ta jest wykorzystana zamiast pojedynczej instancji klasy `Random`, aby uniknąć opóźnień związanych z synchronizacją dostępu do jej metody `next()`. Dzięki zmiennym lokalnym problem synchronizacji nie istnieje, ponieważ każdy wątek ma nieograniczony dostęp do swojego obiektu.

W tym przykładzie problem z synchronizacją można byłoby łatwo rozwiązać tworząc instancję klasy `Random` za każdym razem, kiedy byłaby potrzebna. Jednak nie poprawiłoby to wydajności aplikacji. Inicjowanie obiektu typu `Random` jest czasochłonne, a ciągłe powtarzanie tej operacji zajmowałoby w sumie więcej czasu niż synchronizowanie dostępu dla wielu wątków do pojedynczej instancji tej klasy.

Lepszy rezultat daje użycie klasy `ThreadLocalRandom`, co potwierdza tabela 7.3 przedstawiająca czasy wygenerowania 10 000 losowych liczb przez każdy z czterech wątków. Wyniki obejmują następujące przypadki:

- tworzenie obiektu typu `Random` przez każdy wątek,
- współdzielenie przez wątki pojedynczego statycznego obiektu typu `Random`,
- współdzielenie przez wątki pojedynczego statycznego obiektu typu `ThreadLocalRandom`.

Tabela 7.3. Efekt użycia klasy ThreadLocalRandom do wygenerowania 10 000 liczb losowych

Przypadek	Czas trwania
Tworzenie obiektów typu `Random`	134,9 ± 0,01 ms
Współdzielenie obiektu typu `Random`	3763 ± 200 ms
Współdzielenie obiektu typu `ThreadLocalRandom`	52,0 ± 0,01 ms

Mikrotesty aplikacji, w których wątki blokują dostęp do zasobów, nigdy nie są wiarygodne. W drugim opisywanym przypadku wątki niemal zawsze musiały czekać na zwolnienie blokady obiektu `Random`. W rzeczywistej aplikacji skala rywalizacji wątków na pewno byłaby znacznie mniejsza. Niemniej jednak w przypadku stosowania współdzielonego obiektu należy spodziewać się pewnej rywalizacji pomiędzy wątkami. Natomiast tworzenie obiektu typu `Random` trwa ponad dwukrotnie dłużej niż uzyskanie dostępu do obiektu typu `ThreadLocalRandom`.

Z opisanych przykładów płynie taki wniosek, że jeżeli inicjowanie obiektów trwa długo, można śmiało tworzyć ich pule lub stosować lokalne zmienne wątkowe, aby móc je wielokrotnie wykorzystywać. Jak zawsze jednak należy szukać właściwego kompromisu, ponieważ zbyt duża pula podstawowych obiektów niemal na pewno pogłębi problemy z wydajnością aplikacji, które z założenia miała rozwiązać. Opisane techniki należy stosować w przypadku obiektów, których inicjowanie jest czasochłonne, a liczba obiektów wielokrotnego użytku jest niewielka.

Krótkie podsumowanie

- Zazwyczaj należy unikać stosowania obiektów wielokrotnego użytku. Ta technika jest uzasadniona, jeżeli wykorzystywana jest niewielka grupa obiektów, a ich inicjowanie jest czasochłonne.
- Pule obiektów i lokalne zmienne wątkowe mają swoje zalety i wady. Na ogół zmienne są łatwiejsze w użyciu, pod warunkiem, że każdy wątek może wykorzystywać własną zmienną.

Odwołania miękkie, słabe i inne

Dzięki miękkim i słabym odwołaniom można wielokrotnie wykorzystywać obiekty, choć zazwyczaj programiści nie myślą w tych kategoriach. Tego rodzaju odwołania, które będę nazywał **niezdefiniowanymi** (ang. *indefinite references*), są częściej wykorzystywane do zapisywania wyników czasochłonnych obliczeń lub danych odczytanych z bazy, niż do tworzenia prostych obiektów wielokrotnego użytku. Na przykład w serwerze danych giełdowych niezdefiniowane odwołanie może być wykorzystywane do przechowywania wyniku zwróconego przez metodę `getHistory()`, która wykonuje długotrwałe obliczenia i wysyła do bazy danych czasochłonne zapytania. Wynikiem zwracanym przez tę metodę jest obiekt, którego inicjowanie jest czasochłonne, ale dzięki niezdefiniowanemu odwołaniu można go wykorzystywać wielokrotnie.

Programiści jednak różnie rozumieją powyższe pojęcia, mimo że oddają one istotę sprawy. Nikt nie mówi o „zapamiętywaniu" wątku w celu jego ponownego użycia, natomiast w tym podrozdziale będzie mowa o niezdefiniowanych odwołaniach wielokrotnego użytku zapamiętujących wyniki operacji wykonywanych na bazie danych.

Uwaga do stosowanej terminologii

Terminy „miękkie odwołania” i „słabe odwołania” są do siebie podobne, więc mogą się mylić. Dlatego poniżej znajduje się krótki glosariusz:

Odwołanie

Dowolnego rodzaju odwołanie do obiektu — silne, słabe, miękkie itp. Zwykła zmienna instancji zawierająca obiekt jest silnym odwołaniem.

Niezdefiniowane odwołanie

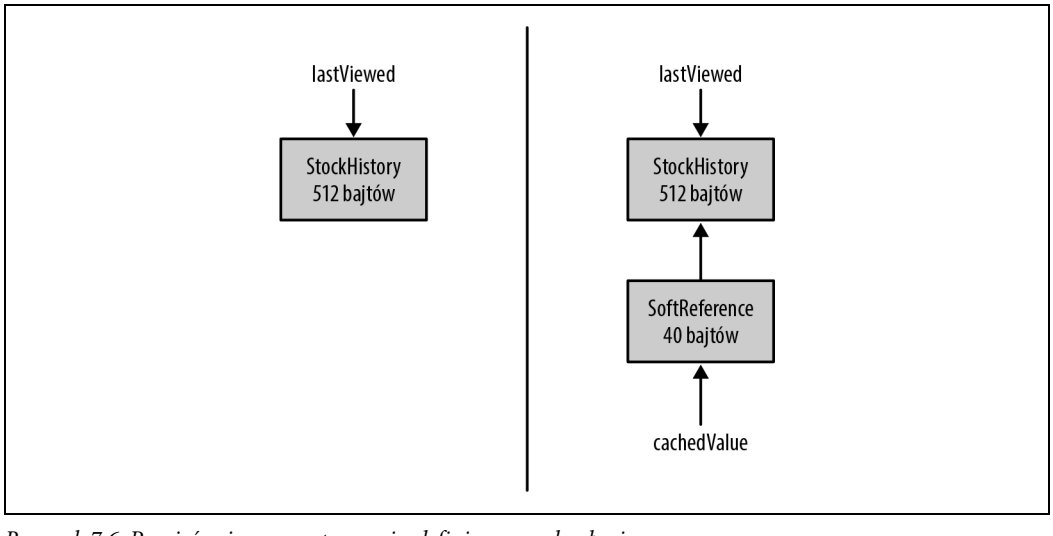
Tego terminu używam na określenie odwołań specjalnego rodzaju, m.in. miękkich i słabych. Oznacza obiekt, np. instancję klasy `SoftReference`.

Referent

Niezdefiniowane odwołanie jest zazwyczaj opakowaniem innego odwołania, niemal zawsze silnego. Opakowujący obiekt jest nazywany referentem.

Niezdefiniowane odwołanie ma tę przewagę nad pulą obiektów czy lokalną zmienną wątkową, że może zostać całkowicie usunięte przez kolektor. Jeżeli w puli obiektów umieści się wyniki 10 000 zapytań o akcje i na stercie zacznie brakować miejsca, pojawi się problem. Aplikacja będzie mogła wykorzystywać tylko pozostałą część sterty. Jeżeli wyniki będą przechowywane w niezdefiniowanych odwołaniach, kolektor zwolni pewną ilość pamięci, zależną od odwołania, dzięki czemu porządkowanie pamięci będzie bardziej wydajne.

Wada niezdefiniowanych odwołań polega na tym, że nieco pogarszają wydajność kolektora. Rysunek 7.6 przedstawia porównanie dwóch przypadków. W pierwszym nie są wykorzystywane niezdefiniowane odwołania (tutaj miękkie), a w drugim są.



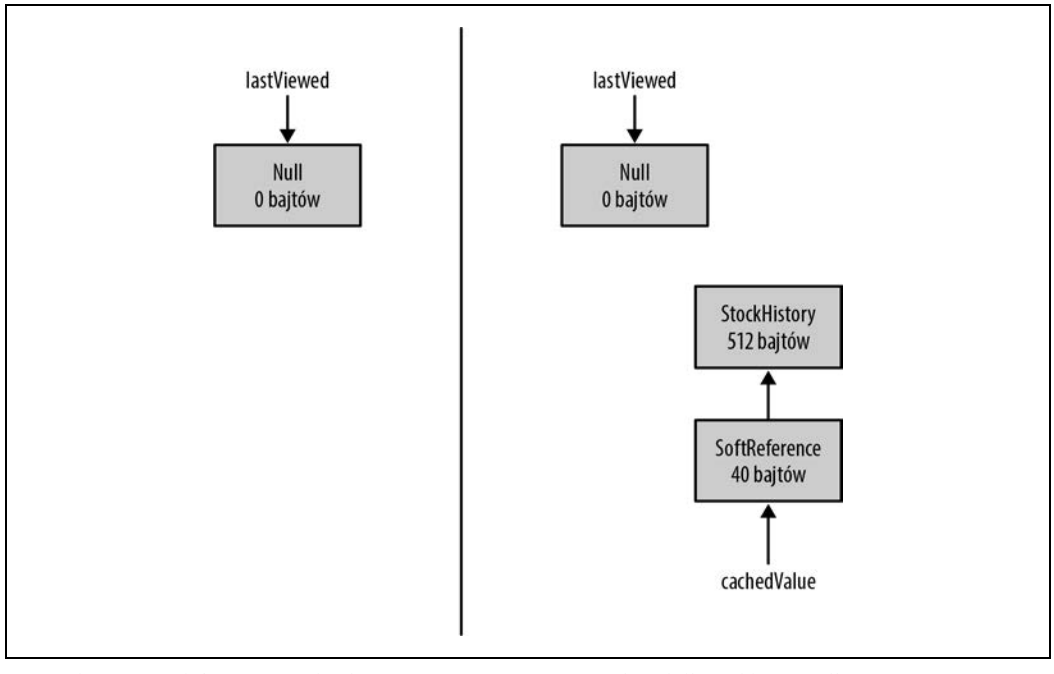
Rysunek 7.6. Pamięć zajmowana przez niezdefiniowane odwołanie

Zapisany w pamięci obiekt zajmuje 512 bajtów. Rysunek po lewej stronie przedstawia całkowitą zajmowaną przez niego pamięć (przy czym nie jest uwzględniona zmienna instancji odwołująca się

do tego obiektu). Po prawej stronie przedstawiony jest obiekt umieszczony wewnątrz obiektu typu `SoftReference`, który zajmuje dodatkowe 40 bajtów pamięci. Niezdefiniowane odwołanie jest obiektem takim samym, jak każdy inny, tj. zajmuje pamięć, a odwołująca się do niego zmienna, w tym przypadku `cachedValue`, jest silnym odwołaniem.

Zatem negatywny wpływ niezdefiniowanych odwołań na kolektor polega przede wszystkim na tym, że zajmują one więcej pamięci. Druga, poważniejsza wada, jest taka, że kolektor w celu całkowitego usunięcia niezdefiniowanego odwołania musi wykonać dwa cykle porządkowania pamięci.

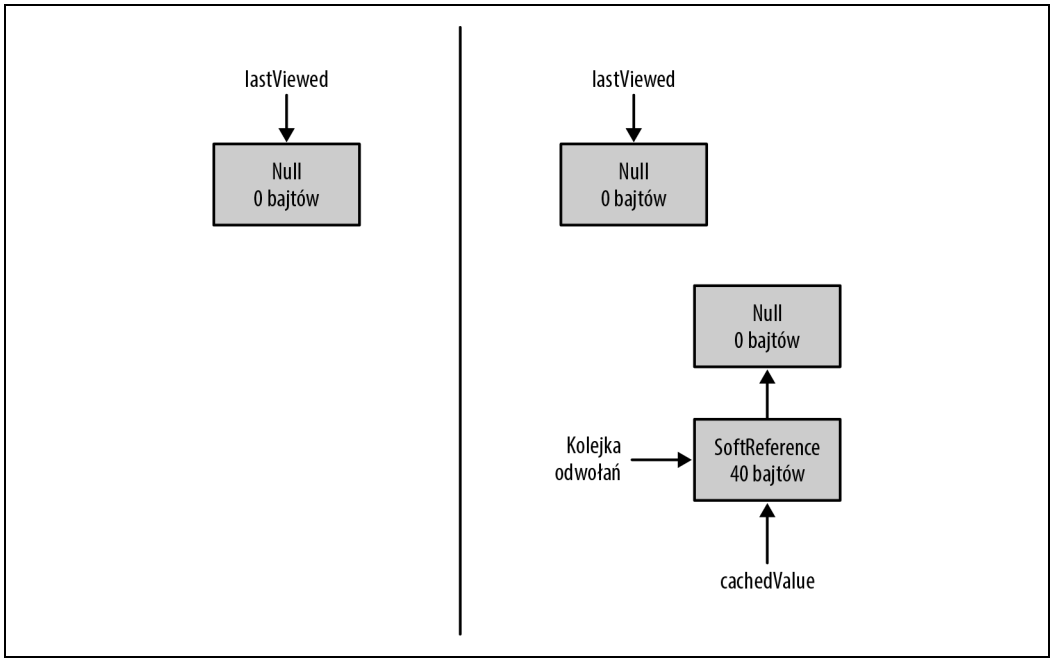
Rysunek 7.7 pokazuje, co się stanie, gdy zniknie silne odwołanie do referenta, tj. gdy zmiennej `lastViewed` zostanie przypisana wartość `null`. W takim wypadku obiekt typu `StockHistory` zostanie usunięty podczas najbliższego porządkowania obszaru sterty, w którym się znajduje. W przypadku takim, jak po lewej stronie, zajętość pamięci jest równa 0 bajtów.



Rysunek 7.7. Niezdefiniowane odwołanie zajmuje pamięć przez okres kilku cykli porządkowania pamięci

Natomiast w sytuacji pokazanej po prawej stronie rysunku pamięć wciąż jest zajmowana. Moment, w którym zostanie usunięty referent, zależy od rodzaju niezdefiniowanego odwołania. Na razie przyjmijmy, że jest to miękkie odwołanie. Referent będzie istniał dotąd, aż maszyna JVM uzna, że jest on wykorzystywany za długo. Wtedy w pierwszym cyklu porządkowania pamięci usunie referenta, ale nie obiekt z miękkim odwołaniem. W efekcie pamięć będzie wyglądała tak, jak na rysunku 7.8.

W tym momencie istnieją przynajmniej dwa silne odwołania do obiektu: oryginalne, utworzone przez aplikację, oraz nowe, utworzone przez maszynę JVM i umieszczone w kolejce odwołań. Wszystkie silne odwołania muszą zostać usunięte, zanim obiekt zawierający niezdefiniowane odwołanie zostanie usunięty przez kolektor.



Rysunek 7.8. Niezdefiniowane odwołanie nie jest usuwane natychmiast

Zazwyczaj kolejkę odwołań oczyszcza kod, który wykonuje na niej operacje. Wysyłany jest do niego sygnał, że w kolejce został umieszczony nowy obiekt i wszystkie silne odwołania do obiektu są natychmiast usuwane. Dzięki temu podczas kolejnego cyklu porządkowania pamięci może zostać usunięty referent. W najgorszym przypadku kolejka odwołań nie jest przetwarzana natychmiast i w efekcie jej całkowite oczyszczenie może zająć kilka cykli porządkowania. Jednak nawet w najkorzystniejszym przypadku niezdefiniowane odwołanie jest usuwane w dwóch cyklach.

Powyższy ogólny algorytm może wyglądać inaczej w zależności od rodzaju niezdefiniowanych odwołań, niemniej jednak wszystkie w podobny sposób pogarszają wydajność aplikacji.

Dziennik kolektora i obsługa odwołań

Jeżeli aplikacja wykorzystuje dużo niezdefiniowanych odwołań, warto użyć flagi `-XX:+PrintReferenceGC`, która domyślnie ma wartość `true`. Dzięki niej można sprawdzić, ile czasu trwa przetwarzanie tego rodzaju odwołań. Ilustruje to poniższy przykład:

```
[GC[SoftReference, 0 refs, 0.0000060 secs]
        [WeakReference, 238425 refs, 0.0236510 secs]
        [FinalReference, 4 refs, 0.0000160 secs]
        [PhantomReference, 0 refs, 0.0000010 secs]
        [JNI Weak Reference, 0.0000020 secs]
        [PSYoungGen: 271630K->17566K(305856K)]
        271630K->17566K(1004928K), 0.0797140 secs]
        [Times: user=0.16 sys=0.01, real=0.08 secs]
```

W tym przypadku z powodu 238 425 słabych odwołań porządkowanie obszaru młodej generacji trwało 23 ms dłużej.

Miękkie odwołania

Miękkie odwołania stosuje się wtedy, gdy dany obiekt z dużym prawdopodobieństwem będzie wielokrotnie wykorzystywany w przyszłości, ale kolektor powinien go usuwać, gdy nie będzie używany przez jakiś czas (przy podejmowaniu decyzji brana jest również pod uwagę ilość wolnego miejsca na stercie). Miękkie odwołania to w rzeczywistości jedna duża pula obiektów LRU (ang. *least recently used* — ostatnio używane). Kluczem do uzyskania wysokiej wydajności puli jest jej regularne oczyszczanie.

Poniżej opisany jest przykład. Serwer danych giełdowych może zawierać globalną pamięć podręczną przechowującą historię akcji. Kluczami mogą być symbole akcji lub symbole z datami. Gdy serwer odbierze zapytanie o historię akcji `TPKS` z okresu od 1.09.2019 do 31.12.2019, najpierw sprawdzi, czy w pamięci podręcznej zostały wcześniej zapisane wyniki podobnego zapytania.

Dane są umieszczane w pamięci podręcznej, ponieważ zapytania o niektóre z nich pojawiają się częściej niż o inne. Jeżeli najwięcej dotychczasowych zapytań dotyczyło akcji `TPKS`, można się spodziewać, że ich dane zostaną umieszczone w pamięci podręcznej w postaci miękkich odwołań. Z drugiej strony, pojedyncze zapytanie o akcje `KENG` spowoduje umieszczenie w pamięci danych, które za jakiś czas zostaną z niej usunięte. Podobna zasada dotyczy kolejnych zapytań. Jeżeli nadejdzie seria zapytań o akcje `DNLD`, będzie można wykorzystać wyniki zapisane dla pierwszego zapytania. Jeżeli użytkownicy uznają, że powyższe akcje są złą inwestycją, dane po jakimś czasie zostaną usunięte z pamięci.

Kiedy dokładnie są usuwane miękkie odwołania? Najpierw muszą zostać usunięte wszystkie silne odwołania do referenta. Gdy pozostanie tylko miękkie odwołanie, referent jest usuwany podczas następnego porządkowania pamięci (pod warunkiem, że odwołanie nie było niedawno wykorzystywane). Algorytm można opisać następującym pseudokodem:

```
long ms = SoftRefLRUPolicyMSPerMB * AmountOfFreeMemoryInMB;
if (teraz – czas_ostatniego_użycia_odwołania > ms)
   usunięcie_odwołania
```

Powyższy kod wykorzystuje dwie wartości. Pierwszą z nich określa się za pomocą flagi `-XX:Soft` ↪`RefLRUPolicyMSPerMB=`*N* o domyślnej wartości 1000. Drugą wartością jest wielkość wolnego miejsca na stercie po zakończeniu cyklu porządkowania. Ilość ta jest różnicą pomiędzy maksymalną wielkością sterty a zajętym miejscem.

Jak to wszystko działa? Przeanalizujmy przykład maszyny JVM ze stertą o wielkości 4 GB. Sterta po pełnym (współbieżnym) uporządkowaniu może być wypełniona w 50%. Zatem wolne miejsce ma wielkość 2 GB. Domyślna wartość flagi `SoftRefLRUPolicyMSPerMB` (1000) oznacza, że wszystkie miękkie odwołania, które nie były wykorzystywane przez ostatnie 2048 sekund (2 048 000 ms), mają być usuwane. Wielkość wolnego miejsca, tj. 2048 MB, jest mnożona przez 1000:

```
long ms = 2048000; // 1000 * 2048
if (System.currentTimeMillis() - czas_ostatniego_użycia_odwołania > ms)
    usunięcie_odwołania
```

Gdyby sterta była zapełniona w 75%, usuwane byłyby odwołania, które nie były wykorzystywane przez co najmniej 1024 sekundy itd.

Aby zwiększyć częstość usuwania miękkich odwołań, należy zmniejszyć wartość flagi `SoftRefLRU↪PolicyMSPerMB`. Przypisanie fladze wartości 500 oznacza, że ze sterty o wielkości 4 GB, zapełnionej w 75%, będą usuwane odwołania niewykorzystywane co najmniej przez 512 sekund.

Zmiana wartości tej flagi jest zazwyczaj konieczna w sytuacji, gdy sterta szybko zapełnia się miękkimi odwołaniami. Załóżmy, że na stercie jest 2 GB wolnego miejsca i aplikacja zaczyna tworzyć miękkie odwołania. Jeżeli utworzy odwołania o łącznej wielkości 1,7 GB w czasie krótszym niż 2048 sekund (ok. 34 minuty), żadne z nich nie zostanie usunięte. W efekcie dla innych obiektów pozostanie tylko 300 MB wolnego miejsca i będzie często wykonywane porządkowanie pamięci, co negatywnie wpłynie na wydajność aplikacji.

Jeżeli maszyna JVM zajmie całą pamięć lub zacznie bardzo często porządkować stertę, usunie wszystkie miękkie odwołania, aby uniknąć zgłoszenia błędu `OutOfMemoryError`. Jest to dobre podejście, ale bezkrytyczne usuwanie wyników zapisanych w pamięci podręcznej to gorsza sprawa. Zatem kolejna sytuacja, w której warto zmniejszyć wartość flagi `SoftRefLRUPolicyMSPerMB`, ma miejsce wtedy, gdy w dzienniku kolektora pojawia się wpis o nieoczekiwanym usunięciu dużej liczby miękkich odwołań. Jak wspomniałem w podrozdziale „Zbyt długie porządkowanie pamięci", taki przypadek ma miejsce dopiero po czterech kolejnych cyklach pełnego porządkowania pamięci i spełnieniu dodatkowych warunków.

Z drugiej strony, wartość powyższej flagi można zwiększyć, jeżeli aplikacja jest przeznaczona do długotrwałego działania i spełnione są dwa warunki:

- na stercie jest dużo wolnego miejsca,
- miękkie odwołania są rzadko wykorzystywane.

Jest to nietypowa sytuacja, podobna do opisanej w podrozdziale poświęconym ustawieniom kolektora. Mogłoby się wydawać, że po zwiększeniu wartości flagi usuwanie miękkich odwołań będzie ostatecznością. To prawda, jednak oznacza to również, że na stercie będzie mniej miejsca dla zwykłych obiektów i w efekcie porządkowanie pamięci będzie odbywało się zbyt często.

Należy więc uważać, aby nie tworzyć zbyt wielu miękkich odwołań, ponieważ mogą one szybko zapełnić całą stertę. Jest to nawet ważniejsze niż unikanie tworzenia dużych pul obiektów. Miękkie odwołania sprawdzają się w sytuacji, gdy liczba obiektów jest niewielka. Jeżeli tak nie jest, należy stosować tradycyjną pulę o stałej wielkości, zaimplementowaną jako pamięć podręczna LRU.

Słabe odwołania

Słabe odwołania należy stosować wtedy, gdy dany referent jest wykorzystywany w kilku wątkach jednocześnie. Jeżeli ten warunek nie będzie spełniony, odwołania z dużym prawdopodobieństwem będą usuwane przez kolektor, ponieważ obiekty, do których prowadzą wyłącznie słabe odwołania, są usuwane w każdym cyklu porządkowania pamięci. Oznacza to, że słabe odwołania nigdy nie osiągają stanu właściwego miękkim odwołaniom, przedstawionego na rysunku 7.7. Wraz z silnym odwołaniem natychmiast jest usuwane odwołanie słabe. Zatem program przechodzi ze stanu pokazanego na rysunku 7.6 bezpośrednio do przedstawionego na rysunku 7.8.

Ciekawostką jest miejsce na stercie zajmowane przez słabe odwołania. Odwołania są traktowane tak samo, jak wszystkie obiekty, tj. są tworzone w obszarze młodej generacji i ostatecznie przenoszone do starej. Gdy usuwany jest referent, jego słabe odwołanie znajdujące się w obszarze młodej

generacji jest usuwane bardzo szybko, podczas najbliższego małego porządkowania pamięci (przy założeniu, że kolejka odwołań do danego obiektu jest szybko obsługiwana). Jeżeli referent istnieje długo, jego słabe odwołanie jest przenoszone do starej generacji i usuwane dopiero podczas najbliższego cyklu współbieżnego lub pełnego porządkowania pamięci.

Kontynuujmy przykład pamięci podręcznej wykorzystywanej przez serwer danych giełdowych. Załóżmy, że każdy klient żądający danych o akcjach `TPKS` z dużym prawdopodobieństwem będzie o nie pytał ponownie. Warto więc przechowywać te dane w pamięci podręcznej w postaci silnych odwołań związanych z sesją klienta. Dzięki temu dane te będą dla klienta zawsze dostępne. Gdy klient się wyloguje, sesja zostanie zamknięta, a zajmowana pamięć odzyskana.

Załóżmy teraz, że inny klient zapytuje o akcje `TPKS`. Jak można je znaleźć? Ponieważ obiekt znajduje się gdzieś w pamięci, nie trzeba go ponownie szukać, ale dane skojarzone z sesją nie są dostępne dla innych klientów. Zatem, oprócz silnych odwołań do danych akcji `TPKS` skojarzonych z sesją, warto tworzyć odwołania słabe do danych znajdujących się w głównej pamięci podręcznej. Dzięki temu drugi klient będzie mógł znaleźć te dane, o ile pierwszy nie zamknie sesji. Jest to sytuacja opisana w podrozdziale „Analiza sterty"; dane posiadały tam dwa odwołania i trudno było je znaleźć na podstawie obiektów zajmujących najwięcej zachowanej pamięci.

Na tym właśnie polega jednoczesny dostęp. Można go porównać do następującego polecenia wydawanego maszynie JVM: „Dopóki ktoś jest zainteresowany tym obiektem, mów mi, gdzie on jest. Gdy już nikt tego obiektu nie będzie potrzebował, usuń go, a ja go ponownie sam utworzę". Porównajmy to z miękkim odwołaniem, które nakazuje: „Trzymaj ten obiekt, dopóki jest dostępna wolna pamięć i ktoś może od czasu do czasu korzystać z tego obiektu". Niezrozumienie tej różnicy jest najczęstszą przyczyną problemów z wydajnością aplikacji wykorzystującej słabe odwołania. Błędne jest twierdzenie, że słabe odwołania różnią się od miękkich tylko tym, że są szybciej usuwane. Obiekt z miękkim odwołaniem zazwyczaj istnieje kilka minut (najwyżej godzin), natomiast obiekt ze słabym odwołaniem jest dostępny dopóki istnieje jego referent i jest usuwany w kolejnym cyklu porządkowania pamięci.

Finalizery i odwołania finalne

Każda klasa w Javie posiada metodę `finalize()` odziedziczoną po klasie `Object`. Metodę tę można wykorzystywać do usuwania niepotrzebnych danych, gdy obiekt zostanie zakwalifikowany do usunięcia. Funkcjonalność ta wydaje się ciekawa i wykorzystuje się ją w kilku szczególnych przypadkach. W praktyce okazuje się jednak, że należy zdecydowanie unikać stosowania tej metody.

Finalizery okazały się tak złą funkcjonalnością, że w pakiecie JDK 11 (ale nie w JDK 8) odradza się ich stosowanie. W pozostałej części rozdziału opiszę, dlaczego tak się dzieje, ale teraz przedstawię przyczyny powstania finalizerów.

Finalizery zostały wprowadzone w celu rozwiązania kilku problemów ujawniających się podczas zarządzania cyklem życia obiektów przez maszynę JVM. W językach takich jak C++, w których trzeba jawnie usuwać obiekt, gdy przestaje być potrzebny, można za pomocą destruktora czyścić jego stan. Natomiast w Javie obiekt jest usuwany automatycznie, gdy wyjdzie poza zakres widoczności. W takim wypadku rolę destruktora pełni finalizer.

Niezdefiniowane odwołania i kolekcje

Klasy kolejki są częstym źródłem wycieku pamięci. Jeżeli na przykład aplikacja umieszcza obiekty w mapie `HashMap` i nie usuwa ich, wtedy w miarę upływu czasu mapa rośnie i zajmuje coraz więcej miejsca na stercie.

Jedno z rozwiązań tego problemu polega na stosowaniu klasy kolekcji zawierającej niezdefiniowane odwołania. Pakiet JDK oferuje dwie takie klasy: `WeakHashMap` i `WeakIdentityMap`. W zewnętrznych źródłach są dostępne niestandardowe klasy kolekcji, oparte na miękkich odwołaniach i nie tylko. Są to m.in. własne implementacje będące odpowiedzią na żądane JSR 166, wykorzystane w przykładzie pokazującym, jak tworzyć i przechowywać obiekty kanoniczne.

Powyższe klasy są wygodne w użyciu, jednak należy pamiętać o dwóch efektach ubocznych. Po pierwsze, jak już wcześniej wspomniałem, niezdefiniowane odwołania mogą negatywnie wpływać na działanie kolektora. Po drugie, sama klasa musi od czasu do czasu usuwać z kolekcji dane bez odwołań. Oznacza to, że klasa jest odpowiedzialna za zarządzanie własną kolejką niezdefiniowanych odwołań.

Na przykład klasa `WeakHashMap` wykorzystuje miękkie odwołania jako klucze. Jeżeli klucz przestanie być potrzebny, klasa musi usunąć z mapy skojarzoną z nim wartość. Tę operację musi wykonywać przy każdym odwołaniu do mapy. Kolejka odwołań dla słabego klucza jest przetwarzana, a wartość skojarzona z tym kluczem w kolejce odwołań jest usuwana z mapy.

Wpływ powyższych operacji na wydajność aplikacji jest dwojaki. Po pierwsze, słabe odwołanie i skojarzona z nim wartość nie mogą zostać usunięte do chwili ponownego użycia mapy. Jeżeli więc mapa jest wykorzystywana rzadko, pamięć zajmowana przez mapę nie będzie zwalniana tak szybko, jak powinna być. Po drugie, trudno jest określić wydajność operacji wykonywanych na mapie. Zazwyczaj są one szybkie, dzięki czemu mapy są tak popularne. Podczas operacji wykonywanej na klasie `WeakHashMap` zaraz po uporządkowaniu pamięci musi być przetworzona kolejka odwołań, co jednak nie zajmuje krótkiego, ściśle określonego czasu. Zatem nawet jeżeli klucze są usuwane rzadko, trudno jest przewidzieć, jaka będzie wydajność. Jeżeli natomiast będą usuwane często, wydajność operacji wykonywanych na klasie `WeakHashMap` będzie na pewno bardzo niska.

Kolekcje oparte na niezdefiniowanych odwołaniach mogą być przydatne, ale należy je stosować ostrożnie. Jeżeli jest to możliwe, aplikacja sama musi zarządzać kolekcją.

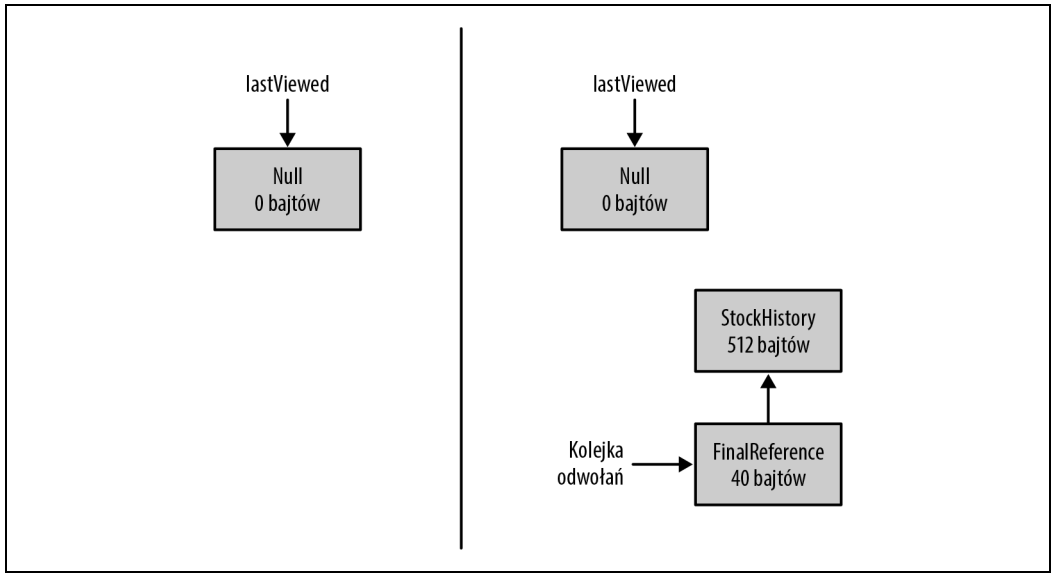
Na przykład w pakiecie JDK finalizery są stosowane do przetwarzania plików ZIP, ponieważ wykorzystywane są wtedy natywny kod i natywna pamięć. Pamięć jest zwalniana, gdy plik jest zamykany. Co się jednak stanie, gdy programista zapomni użyć metody `close()`? Można ją wywołać za pomocą finalizera.

Wiele klas zawartych w pakiecie JDK 8 wykorzystuje finalizery w opisany wyżej sposób, jednak w JDK 11 stosowana jest zamiast nich zupełnie inna funkcjonalność. Jest to klasa `Cleaner`, opisana w następnej części rozdziału. Jeżeli zamierzasz stosować finalizery we własnych klasach lub używasz pakietu JDK 8, w którym powyższa klasa nie jest dostępna, kontynuuj lekturę tego rozdziału, aby dowiedzieć się, jakie są możliwości rozwiązania problemu.

Finalizery są złe pod względem funkcjonalnym i wydajnościowym. Są to szczególnego rodzaju niezdefiniowane odwołania. Maszyna JVM wykorzystuje prywatną klasę odwołania `java.lang.ref.Finalizer`, która z kolei jest rozszerzeniem klasy `java.lang.ref.FinalReference` służącej do rejestrowania obiektów posiadających metodę `finalize()`. Gdy taki obiekt jest tworzony, maszyna

alokuje jeszcze jeden obiekt typu `Finalizer`, dla którego oryginalny obiekt jest referentem. Obiekt ten, tak jak każde niezdefiniowane odwołanie, jest usuwany przynajmniej w dwóch cyklach porządkowania pamięci. Jednak efekty uboczne są wtedy o wiele bardziej dotkliwe niż w przypadku zwykłych niezdefiniowanych odwołań. Referent z miękkim lub słabym odwołaniem jest podczas porządkowania pamięci usuwany natychmiast, przez co zajętość pamięci wygląda tak jak na opisanym wcześniej rysunku 7.8. Miękkie lub słabe odwołanie jest umieszczane w kolejce odwołań, natomiast sam obiekt nie odwołuje się już do niczego (tj. jego metoda `get()` zwraca wartość `null`, a nie oryginalnego referenta). W przypadku miękkich i słabych odwołań dwa cykle porządkowania pamięci dotyczą samego obiektu odwołania, a nie referenta.

Tak jednak nie jest w przypadku odwołań finalnych. Implementacja klasy `Finalizer` musi mieć dostęp do referenta, aby mogła wywołać jego metodę `finalize()`. Zatem nie można usunąć referenta, gdy odwołanie finalne jest umieszczone w jego kolejce odwołań. Gdy referenta można już usunąć, program przyjmuje stan przedstawiony na rysunku 7.9.



Rysunek 7.9. Odwołania finalne zajmują więcej pamięci

Gdy kolejka odwołań uruchamia kod finalizera, obiekt `Finalizer` jest usuwany w zwykły sposób z kolejki, a następnie ze sterty. Dopiero potem usuwany jest referent. Dlatego właśnie finalizery znacznie bardziej obniżają wydajność kolektora niż inne niezdefiniowane odwołania. Referent może zajmować o wiele większą pamięć niż niezdefiniowane odwołanie.

Funkcjonalny problem z finalizerami polega na tym, że metoda `finalize()` może niezauważalnie utworzyć nowe silne odwołanie do referenta, co również pogarsza wydajność kolektora. W takim wypadku referent nie może być usunięty, dopóki nie zniknie silne odwołanie. Problem jest poważny, ponieważ następnym razem, gdy referenta będzie można usunąć, jego metoda `finalize()` nie zostanie wywołana i oczekiwane oczyszczenie referenta nie będzie miało miejsca. Z powodu tego mankamentu należy w miarę możliwości unikać stosowania finalizerów.

Należy więc przestrzegać zasady, że jeżeli nie da się uniknąć użycia finalizera, trzeba ograniczyć do minimum pamięć zajmowaną przez obiekt.

Istnieje jednak alternatywa dla finalizerów, dzięki której można uniknąć przynajmniej części problemów, a dodatkowo usuwać referenta podczas zwykłego porządkowania pamięci. W tym celu, zamiast niejawnie stosować klasę `Finalizer`, wystarczy po prostu użyć innego rodzaju niezdefiniowanego odwołania.

Czasami zalecane jest stosowanie jeszcze innego rodzaju niezdefiniowanego odwołania — klasy `PhantomReference`. W rzeczywistości klasa ta jest stosowana w pakiecie JDK 11. Obiekt `Cleaner` jest znacznie prostszy w użyciu niż w opisanym przykładzie, który w rzeczywistości dotyczy tylko pakietu JDK 8. Jest to dobry wybór, ponieważ obiekt odwołania jest wtedy usuwany dość szybko, gdy znikną silne odwołania do referenta. Przeznaczenie odwołania podczas diagnozowania kodu jest oczywiste. Ten sam cel można osiągnąć za pomocą słabych odwołań (które mogą być stosowane w innych miejscach). W szczególnych sytuacjach można zastosować miękkie odwołania, jeśli ich buforowanie odpowiada potrzebom aplikacji.

Aby utworzyć zamiennik finalizera, należy zdefiniować podklasę niezdefiniowanego odwołania. Wszelkie umieszczone w niej dane będą usuwane po usunięciu referenta. Następnie trzeba zdefiniować metodę oczyszczającą obiekt odwołania, zamiast metody `finalize()` w klasie referenta.

Poniżej przedstawiona jest struktura takiej klasy, wykorzystującej miękkie odwołanie. Jej konstruktor alokuje natywne zasoby. Podczas zwykłego korzystania z klasy będzie wywoływana metoda `setClosed()` oczyszczająca natywną pamięć.

```
private static class CleanupFinalizer extends WeakReference {
  private static ReferenceQueue<CleanupFinalizer> finRefQueue;
  private static HashSet<CleanupFinalizer> pendingRefs = new HashSet<>();
  private boolean closed = false;

  public CleanupFinalizer(Object o) {
    super(o, finRefQueue);
    allocateNative();
    pendingRefs.add(this);
  }

  public void setClosed() {
    closed = true;
    doNativeCleanup();
  }

  public void cleanup() {
    if (!closed) {
      doNativeCleanup();
    }
  }

  private native void allocateNative();
  private native void doNativeCleanup();
}
```

Jednak słabe odwołanie również jest umieszczane w kolejce odwołań. Gdy odwołanie jest pobierane z kolejki, może zwalniać natywną pamięć, jeżeli nie zostało to zrobione wcześniej.

Kolejka odwołań jest przetwarzana przez wątek działający w tle aplikacji:

```
static {
  finRefQueue = new ReferenceQueue<>();
  Runnable r = new Runnable() {
    public void run() {
      CleanupFinalizer fr;
      while (true) {
        try {
          fr = (CleanupFinalizer) finRefQueue.remove();
          fr.cleanup();
          pendingRefs.remove(fr);
        } catch (Exception ex) {
          Logger.getLogger(
            CleanupFinalizer.class.getName()).
              log(Level.SEVERE, null, ex);
        }
      }
    }
  };
  Thread t = new Thread(r);
  t.setDaemon(true);
  t.start();
}
```

Powyższy kod jest prywatną klasą statyczną, niewidoczną dla programisty używającego właściwej klasy, która wygląda następująco:

```
public class CleanupExample {
  private CleanupFinalizer cf;
  private HashMap data = new HashMap();

  public CleanupExample() {
      cf = new CleanupFinalizer(this);
  }

  ...Metody umieszczające dane w tablicy mieszającej...
  public void close() {
    data = null;
    cf.setClosed();
  }
}
```

Programista tworzy obiekt w zwykły sposób i wie, że musi wywołać metodę `close()`, aby zwolnić natywną pamięć. Jednak nic złego się nie stanie, jeżeli tego nie zrobi. Niewidoczne słabe odwołanie będzie wciąż istniało, więc klasa `CleanupFinalizer` zwolni pamięć, gdy wewnętrzna klasa przetworzy słabe odwołanie.

Jedyna trudność w tym przykładzie polega na tym, że zmiennej `pendingRefs` trzeba przypisać słabe odwołanie. Jeżeli się tego nie zrobi, słabe odwołania będą usuwane zanim zostaną umieszczone w kolejce.

W tym przykładzie rozwiązywane są dwa problemy typowe dla tradycyjnego finalizera. Poprawiana jest wydajność aplikacji, ponieważ dane powiązane z referentem (w tym przypadku tablicą mieszającą) są usuwane razem z nim, więc nie trzeba wywoływać metody `finalizer()`. Ponadto zapobiega się przywróceniu obiektu referenta w kodzie oczyszczającym, ponieważ obiekt ten jest wcześniej usuwany.

Powyższy kod wciąż jednak ma mankamenty typowe dla finalizera. Nie daje mianowicie pewności, że kolektor ostatecznie usunie referenta, ani że wątek zarządzający kolejką odwołań kiedykolwiek przetworzy określony, znajdujący się w niej obiekt. Jeżeli takich obiektów jest dużo, przetwarzanie kolejki odwołań jest czasochłonne. Dlatego należy ostrożnie stosować opisany wyżej kod, podobnie jak wszelkiego rodzaju niezdefiniowane odwołania.

Kolejka finalizera

Kolejka finalizera jest to kolejka odwołań wykorzystywana do przetwarzania odwołania `Finalizer`, gdy referent zostanie zakwalifikowany do usunięcia podczas kolejnego porządkowania pamięci.

Podczas analizowania zrzutu sterty warto sprawdzać, czy kolejka finalizerów jest pusta. Umieszczone w niej obiekty i tak zostaną usunięte, ale jeżeli się je usunie, będzie lepiej widać, co się dzieje na stercie. Przetwarzanie kolejki finalizerów inicjuje się za pomocą następującego polecenia:

```
% jcmd process_id GC.run_finalization
```

Jeżeli pojawi się podejrzenie, że przyczyną problemów z aplikacją jest kolejka finalizerów, należy sprawdzić jej długość. Program `jconsole` na bieżąco podaje tę wartość w zakładce *VM Summary* (podsumowanie maszyny wirtualnej). W skrypcie można ją uzyskać za pomocą poniższego polecenia:

```
% jmap -finalizerinfo process_id
```

Klasa Cleaner

Klasa `java.lang.ref.Cleaner`, dostępna w pakiecie JDK 11, jest znacznie prostsza w użyciu niż metoda `finalize()`. Uzyskuje ona za pomocą klasy `PhantomReference` informację, że obiekt z silnym odwołaniem nie jest już używany. Obowiązuje tu ta sama zasada, co w zalecanej wcześniej klasie `Cleanup` ↪`Finalizer` w pakiecie JDK 8. Ponieważ jednak jest to podstawowa funkcjonalność pakietu JDK, programista nie musi zajmować się przetwarzaniem wątków i własnych odwołań. Wystarczy, że zarejestruje odpowiedni obiekt, który powinien zostać przetworzony przez kod oczyszczający, a całą resztą zajmą się główne biblioteki.

Z perspektywy wydajności najtrudniejszym zadaniem jest zarejestrowanie „odpowiedniego" obiektu przez kod oczyszczający. Kod ten będzie utrzymywał silne odwołanie do zarejestrowanego obiektu, więc obiekt ten nigdy nie będzie osiągalny. Zamiast tego należy utworzyć dowolny przesłonięty obiekt i zarejestrować go.

Przeanalizujmy przykład klasy `java.util.zip.Inflater`. Klasa ta wymaga oczyszczania, ponieważ musi zwalniać podczas przetwarzania zajmowaną pamięć. Kod oczyszczający jest wykonywany po wywołaniu metody `end()`. Należy wywoływać tę metodę, gdy obiekt przestanie być potrzebny. Jednak podczas usuwania obiektu trzeba sprawdzać, czy metoda ta została wywołana. W przeciwnym wypadku będzie miał miejsce wyciek pamięci[1].

[1] Jeżeli programista nie zakoduje jawnego wywołania metody `end()` i zda się na oczyszczanie natywnej pamięci przez kolektor, to również wtedy pojawi się wyciek pamięci. Więcej szczegółowych informacji na ten temat znajduje się w rozdziale 8.

Pseudokod klasy `Inflater` wygląda następująco:

```
public class java.util.zip.Inflater {
  private static class InflaterZStreamRef implements Runnable {
    private long addr;
    private final Cleanable cleanable;
    InflaterZStreamRef(Inflater owner, long addr) {
      this.addr = addr;
      cleanable = CleanerFactory.cleaner().register(owner, this);
    }

    void clean() {
      cleanable.clean();
    }

    private static native void freeNativeMemory(long addr);
    public synchronized void run() {
      freeNativeMemory(addr);
    }
  }

  private InflaterZStreamRef zsRef;

  public Inflater() {
    this.zsRef = new InflaterZStreamRef(this, allocateNativeMemory());
  }

  public void end() {
    synchronized(zsRef) {
      zsRef.clean();
    }
  }
}
```

Powyższy kod jest prostszy niż rzeczywista implementacja, która z uwagi na kompatybilność musi rejestrować klasy nadpisujące metodę `end()`. Poza tym, oczywiście, alokacja natywnej pamięci jest bardziej skomplikowana. W tym przykładzie ważne jest, że wewnętrzna klasa zawiera obiekt, do którego klasa `Cleaner` ma silne odwołanie. Argument zewnętrznej klasy (`owner`), który również jest rejestrowany przez kod oczyszczający, zawiera wyzwalacz. Jeżeli istnieje tylko jeden fantom, uruchamiany jest kod oczyszczający i można użyć zapisanego silnego odwołania jako zaczepu do czyszczenia.

Zwróć uwagę, że wewnętrzna klasa musi być statyczna. W przeciwnym wypadku będzie zawierała niejawne odwołanie do klasy `Inflater`, a obiekt `Inflater` nigdy nie będzie dostępny. Zawsze będzie istniało silne odwołanie z obiektu typu `Cleaner` do obiektu `InflaterZStreamRef`, a z kolei z niego silne odwołanie do obiektu `Inflater`. Z reguły obiekt oczyszczający nie może zawierać odwołań do obiektu oczyszczanego. Dlatego nie należy stosować funkcji lambda zamiast klasy, ponieważ funkcja zbył łatwo odwołuje się do opakowującej klasy.

Krótkie podsumowanie

- Odwołania niezdefiniowane (miękkie, słabe, fantomowe i finalne) zmieniają zwykły cykl życia obiektów, które dzięki nim mogą być wielokrotnie wykorzystane i w mniejszym stopniu niż pule obiektów i lokalne zmienne wątkowe zakłócać działanie kolektora.
- Słabe odwołania można stosować wtedy, gdy aplikacja wykorzystuje obiekty z silnymi odwołaniami.
- Miękkie odwołania mogą utrzymywać obiekty przez dłuższy czas i tworzyć prostą pamięć podręczną LRU, która nie zakłóca porządkowania pamięci.
- Niezdefiniowane odwołania zajmują dodatkową pamięć i utrzymują inne obiekty przez dłuższy czas. Dlatego należy je stosować oszczędnie.
- Finalizery to specjalnego rodzaju odwołania zaprojektowane z myślą o oczyszczaniu obiektów. Zamiast nich należy jednak stosować nową klasę `Cleaner`.

Skompresowane wskaźniki

Proste aplikacje działają na 64-bitowej maszynie JVM wolniej niż na 32-bitowej. Przyczyną są 64-bitowe odwołania do obiektów, które zajmują dwa razy więcej miejsca na stercie (8 bajtów) niż odwołania 32-bitowe (4 bajty). Ponieważ z tego powodu jest mniej miejsca na stercie dla innych danych, maszyna częściej porządkuje pamięć.

Zwiększenie zajętości sterty można zniwelować stosując **skompresowane wskaźniki prostych obiektów** (ang. *ordinary object pointers* — OOP). Są to uchwyty, które maszyna JVM wykorzystuje jako odwołania do obiektów. Wskaźniki 32-bitowe można stosować jedynie wtedy, gdy wielkość pamięci nie przekracza 4 GB (2^{32} bajtów). Z tego właśnie powodu 32-bitowa maszyna JVM jest w stanie obsłużyć stertę o maksymalnej wielkości 4 GB. To samo ograniczenie dotyczy całego systemu operacyjnego. System 32-bitowy może być wyposażony w pamięć o maksymalnej wielkości 4 GB. Za pomocą wskaźników 64-bitowych można obsłużyć pamięć rzędu eksabajtów, czyli znacznie większą niż dostępna w jakimkolwiek komputerze.

Czy istnieje jakiś kompromis, na przykład wskaźniki 35-bitowe? Takie wskaźniki mogłyby obsłużyć pamięć o wielkości 32 GB (2^{35} bajtów) i zajmowałyby mniej miejsca na stercie niż wskaźniki 64-bitowe. Problem jednak polega na tym, że procesor nie posiada 35-bitowych rejestrów, w których mógłby zapisywać tego rodzaju odwołania. Zamiast tego maszyna JVM może przyjąć, że ostatnie 3 bity odwołania są równe 0. Dzięki temu każde odwołanie może zajmować 32 bity na stercie. Podczas umieszczania go w 64-bitowym rejestrze maszyna przesuwa je w lewo o 3 i dodaje 3 zera na końcu, a podczas odczytywania z rejestru przesuwa w prawo o tyle samo bitów i usuwa zera.

W ten sposób maszyna JVM może używać wskaźników obsługujących pamięć o wielkości 32 GB i zajmujących 32 bity na stercie. Jednak oznacza to również, że maszyna nie może odwołać się do obiektu, którego adres nie jest wielokrotnością liczby 8, ponieważ adres zawarty w skompresowanym wskaźniku musi kończyć się trzema zerami. Zatem pierwszy możliwy wskaźnik jest równy 0x1; po przesunięciu zmienia się on na 0x8, następny jest równy 0x2, po przesunięciu 0x10 (16) itd. Obiekty muszą więc być wyrównane do 8-bajtowych segmentów pamięci.

Okazuje się, że maszyna JVM już wcześniej wyrównywała obiekty w ten sposób. Dla większości procesorów jest to optymalne wyrównanie. W związku z tym niczego się nie traci stosując skompresowane wskaźniki. Jeżeli obiekt ma wielkość 57 bajtów i zostanie umieszczony na pozycji 0, to następny obiekt będzie umieszczony na pozycji 64. W efekcie 7 bajtów pozostanie niewykorzystanych. Takie marnotrawstwo pamięci, które nie ma nic wspólnego ze skompresowanymi wskaźnikami, jest jednak praktykowane, ponieważ dostęp do obiektów wyrównanych do 8-bajtowych segmentów jest znacznie szybszy.

Z powodu niewykorzystywanej pamięci maszyna JVM nie emuluje wskaźników 36-bitowych, które mogłyby obsłużyć pamięć o wielkości 64 GB. Obiekty musiałyby być wyrównywane do 16-bajtowych segmentów, przez co tracone byłyby oszczędności wynikające ze stosowania skompresowanych wskaźników.

Opisany kompromis ma dwie konsekwencje. Po pierwsze, skompresowane wskaźniki można stosować jeżeli sterta ma wielkość od 4 GB do 32 GB. Włącza się je za pomocą flagi `-XX:+UseCompressedOops`. Tego rodzaju wskaźniki są stosowane domyślnie, jeżeli maksymalna wielkość sterty nie przekracza 32 GB. W podrozdziale „Zmniejszanie wielkości obiektów" wspomniałem, że jeżeli sterta ma wielkość 32 GB i jest wykorzystywana 64-bitowa maszyna JVM, to odwołanie zajmuje 4 bajty. Wynika to z domyślnego stosowania skompresowanych wskaźników.

Po drugie, aplikacja ze stertą o wielkości 31 GB działa szybciej niż ze stertą o wielkości 33 GB. Miejsca jest wprawdzie więcej, jednak wskaźniki zajmują większą część sterty, przez co musi być ona porządkowana częściej, a to pogarsza wydajność aplikacji.

Zatem warto stosować stertę mniejszą niż 32 GB lub najwyżej kilka GB większą od tej wartości. Jeżeli sterta zostanie powiększona na tyle, że zrobi się na niej miejsce dla nieskompresowanych wskaźników, to liczba cykli porządkowania pamięci zmniejszy się. Nie ma uniwersalnej reguły określającej wielkość pamięci, przy której nieskompresowane wskaźniki nie pogorszą wydajności kolektora. Można jednak przyjąć, że musi to być co najmniej 38 GB, z czego 20% zajmowałyby odwołania do obiektów.

Krótkie podsumowanie

- Skompresowane wskaźniki są stosowane domyślnie, jeżeli jest to uzasadnione.
- Wydajność aplikacji wykorzystującej stertę o wielkości 31 GB jest zazwyczaj większa niż w przypadku nieznacznie większej sterty, która jest zbyt duża, aby mogły być stosowane skompresowane wskaźniki.

Podsumowanie

Zarządzanie pamięcią ma zasadniczy wpływ na szybkość działania aplikacji napisanych w Javie. Strojenie kolektora jest ważne, jednak aby aplikacja mogła osiągnąć maksymalną wydajność, musi efektywnie wykorzystywać pamięć.

Do niedawna trendy w rozwoju sprzętu sprawiały, że programiści nie zastanawiali się nad zarządzaniem pamięcią. Skoro mój laptop jest wyposażony w pamięć o wielkości 16 GB, czy ma jakieś znaczenie fakt, że odwołanie do obiektu niepotrzebnie wykorzystuje 8 bajtów? Jednak dzisiaj

w środowiskach chmurowych i kontenerowych z ograniczoną pamięcią ten problem wraca. Nawet gdy aplikacja jest uruchomiona na silnym sprzęcie z dużą stertą, łatwo jest zapomnieć, że zwykły kompromis „czas-przestrzeń" zmienia się w „czas-przestrzeń i czas". Zbyt duża sterta powoduje, że aplikacja działa wolniej, ponieważ musi porządkować większą pamięć. Zarządzanie stertą w Javie jest i zawsze było bardzo ważne.

Zarządzanie pamięcią polega na stosowaniu w odpowiednim czasie odpowiednich funkcjonalności, takich jak pule obiektów, lokalne zmienne wątkowe i niezdefiniowane odwołania. Dzięki nim można zarówno radykalnie zwiększyć wydajność aplikacji, jak i ją pogorszyć, jeżeli stosuje się je nieumiejętnie. Jeżeli liczba obiektów jest niewielka i mniej więcej stała, powyższe funkcjonalności mogą okazać się bardzo przydatne.

Dostrojenie JVM: oto sekret wydajności kodu Javy!

Istnieją dwie strategie rozwiązywania problemów wydajnościowych aplikacji w Javie. Z jednej strony można wykorzystać potężne komputery i przydzielić JVM ogromne zasoby pamięci, z drugiej — w czasach ekspansji rozwiązań opartych na chmurach obliczeniowych nowe znaczenie zyskują małe, jednoprocesorowe komputery. Firmy takie jak Oracle czy Amazon udostępniają tanie serwery, na których można uruchamiać proste aplikacje. Łatwo się przekonać, jak ważne jest właściwe zarządzanie niewielką ilością pamięci w tego rodzaju środowiskach. Każdy, kto programuje w Javie, powinien dokładnie wiedzieć, jak maszyna JVM wykonuje kod i jak należy ją dostrajać, aby osiągała możliwie największą wydajność.

W tej książce opisano wiele funkcjonalności, narzędzi i procedur, dzięki którym można poprawić efektywność kodu napisanego w Javie 8 i 11 LTS. Główny nacisk położono na zagadnienia istotne dla środowisk produkcyjnych, ale przedstawiono również ciekawe nowe technologie, takie jak kompilacja z wyprzedzeniem i eksperymentalne kolektory. Znalazło się tu także omówienie nowości w mechanizmie porządkowania pamięci i rejestratorze Java Flight Recorder, zaprezentowano kwestie funkcjonowania Javy w środowiskach kontenerowych, udoskonalone narzędzie JMH, kompilatory JIT, współdzielone klasy danych, narzędzia do monitorowania wydajności i wiele innych. Publikację tę doceni każdy inżynier zajmujący się JVM, który chce poradzić sobie z nietypowym działaniem systemu, wyciekami pamięci i problemami z jej porządkowaniem.

Najciekawsze zagadnienia:

- platformy i kompilatory Javy a wydajność aplikacji
- porządkowanie pamięci
- zasady testowania wydajności aplikacji
- pakiet JDK i narzędzia do monitorowania aplikacji
- dostrajanie kolektora i interfejsów Java API
- wydajność aplikacji korzystających z baz danych

Scott Oaks jest architektem w Oracle Corporation. Zajmuje się wydajnością chmury Oracle i oprogramowaniem platformy. Wcześniej przez wiele lat pracował w Sun Microsystems — specjalizował się w takich dziedzinach jak: programowanie sieci, technologia RPC czy oprogramowanie OPEN LOOK Virtual Window Manager. W 2001 roku dołączył do Java Performance Group, w której działa aktywnie do dziś. Jest autorem wielu książek dla programistów.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-7031-9
9 788328 370319

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 79,00 zł