27 января 2011 в 21:31

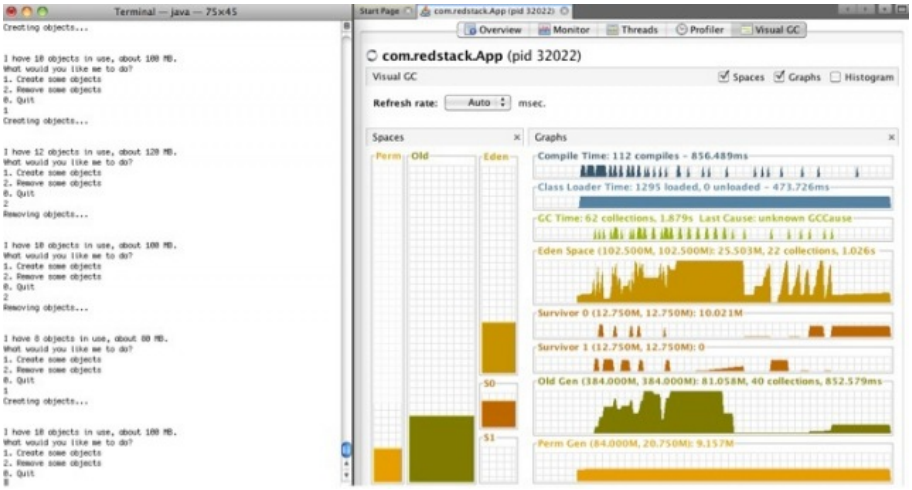
Garbage Collection наглядно

перевод

Java*

Пост посвящён HotSpot JVM – ‘обычной’ JVM от Oracle (раньше Sun), JVM, которую вы скорее всего будете использовать в Windows. В случае Linux это может быть open source JVM. Или JVM может идти в комплекте с другим ПО, например WebLogic, или даже можно использовать Jrockit JVM от Oracle (раньше BEA). Или другие JVM от IBM, Apple и др. Большинство этих «других» JVM работают по похожей на HotSpot схеме, за исключением Jrockit, чье управление памятью отлично от других и который, например, не имеет выделенного Permanent Generation (см. ниже).

Давайте начнём с того, как JVM использует память. В JVM память делится на два сегмента – Heap и Permanent Generation. На диаграмме Permanent Generation обозначено зеленым, остальное – heap.



The Permanent Generation

Permanent generation используется только JVM для хранения необходимых данных, в том числе метаданные о созданных объектах. При каждом создании объекта JVM будет «класть» некоторый набор данных в PG. Соответственно, чем больше вы создаете объектов разных типов, тем больше «жилого пространства» требуется в PG.
Размер PG можно задать двумя параметрами JVM: **-XX:PermSize** – задаёт минимальный, или изначальный, размер PG, и **-XX:MaxPermSize** – задаёт максимальный размер. При запуске больших Java-приложений мы часто задаём одинаковые значения для этих параметров, так что PG создаётся сразу с размером «по-максимуму», что может увеличить производительность, так как изменение размера PG – дорогостоящая (трудоёмкая) операция. Определение одинаковых значений для этих двух параметров может избавить JVM от выполнения дополнительных операций, таких как проверки необходимости изменения размера PG и, естественно, непосредственного изменения.

Heap

Heap – основной сегмент памяти, где хранятся все ваши объекты. Heap делится на два подсегмента, Old Generation и New Generation. New Generation в свою очередь делится на Eden и два сегмента Survivor.
Размер heap также можно указать параметрами. На диаграмме это Xms (минимум) и -Xmx (максимум). Дополнительные параметры контролируют размеры сегментов heap. Мы позднее посмотрим один из них, остальные за рамкой этого поста.
При создании объекта, когда вы пишете что-то типа *byte[] data = new byte[1024]*, этот объект создаётся в сегменте Eden. Новые объекты создаются в Eden. Кроме собственно данных для нашего массива байт здесь есть ещё ссылка (указатель) на эти данные. Дальнейшее объяснение упрощено. Когда вы хотите создать новый объект, но места в Eden уже нет, JVM проводит garbage collection, что значит, что JVM ищет в памяти все объекты, которые более не нужны, и избавляется от них.

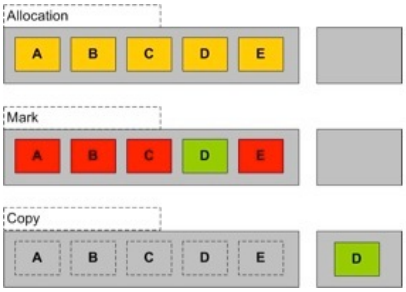
Garbage collection – это круто! Если вы когда-либо программировали на языках типа C или Objective-C, то вы знаете, что ручное управление памятью – вещь утомительная и порой вызывающая ошибки. Наличие JVM, которая автоматически позаботится о неиспользуемых объектах, делает разработку проще и сокращает время отладки. Если же вы никогда не писали на подобных языках, значит возьмите C и попробуйте написать программу, и ощутите, как ценно то, что предоставляется вашим языком совершенно бесплатно.

Существует множество алгоритмов, которыми может воспользоваться JVM для проведения garbage collection. Можно указать, какие

из них будут использоваться JVM с помощью параметров.
Давайте посмотрим на пример. Допустим, у нас есть следующий код:

```
String a = "hello";
String b = "apple";
String c = "banana";
String d = "apricot";
String e = "pear";
//
// делаем чего-нибудь
//
a = null;
b = null;
c = null;
e = null;
```

В Eden создаётся («размещается») пять объектов, как показано пятью желтыми квадратиками на диаграмме. После «чего-нибудь» мы освобождаем a,b,c и e, присваивая ссылкам null. Имея в виду, что больше ссылок на них нет, они теперь не нужны, и показаны красным на второй диаграмме. При этом мы всё ещё нуждаемся в String d (показано зелёным).

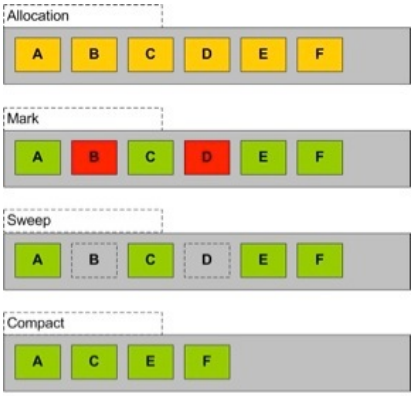


Если мы попробуем разместить ещё объект, JVM обнаружит, что Eden полон и надо провести чистку. Самый простой алгоритм для garbage collection называется Copy Collection, и работает он так, как показано на диаграмме. На первом этапе Mark помечаются неиспользуемые объекты (красные). На втором (Copy) объекты, которые ещё нужны (d) копируется в сегмент survivor – квадрат справа. Сегментов Survivor два, и они меньше Eden. Теперь все объекты, которые мы хотим, чтобы они были сохранены, скопированы в Survivor, и JVM просто удаляет всё из Eden. На этом всё.
Этот алгоритм создаёт кое-что, что называется моментом, «когда мир остановился». Во время выполнения GC все другие треды в JVM переводятся в состояние паузы, ради того, чтобы никакой из них не попробовал влезть в память после того, как мы скопировали всё оттуда, что привело бы к потере того, что сделано. Это не великая проблема, если она в небольшом приложении, но если у нас на руках серьёзная программа, скажем, с 8-гигабайтным heap, то выполнение GC займёт много времени – секунды или даже минуты. Естественно, что каждый раз останавливать приложение – вариант не подходящий. Потому и существуют другие алгоритмы, и используются часто. Copy Collection же работает хорошо в том случае, если у нас много мусора и мало полезных объектов.
В этом посте мы поговорим насчёт двух распространённых алгоритмов. Для интересующихся есть много информации в Сети и несколько хороших книг.
Следующий алгоритм называется Mark-Sweep-Compact Collection. У алгоритма три этапа:

1) «Mark»: помечаются неиспользуемые объекты (красные).

2) «Sweep»: эти объекты удаляются из памяти. Обратите внимание на пустые слоты на диаграмме.

3) «Compact»: объекты размещаются, занимая свободные слоты, что освобождает пространство на тот случай, если потребуется создать «большой» объект.



Но это всё теория, так что давайте посмотрим, как это работает, на примере. К счастью, JDK имеет визуальный инструмент для наблюдения за JVM в реальном времени, а именно jvisualvm. Лежит он прямо в bin JDK. Воспользуемся ею немного позже, сначала займёмся приложением.
Для разработки, билдов и зависимостей я воспользовался Maven, но вам он не нужен – просто скомпилируйте и запустите приложение, если вам так угодно.

```
mvn archetype:generate
  -DarchetypeGroupId=org.apache.maven.archetypes
  -DgroupId=com.redstack
  -DartifactId=memoryTool
```

Я выбрал простой JAR (98) и все по умолчанию для остального. Дальше переключился в директорию memoryTool и отредактировал pom.xml (ниже, добавил блок plugin). Это позволило мне запустить приложение прямо из Maven, передав необходимые параметры.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.redstack</groupId>
  <artifactId>memoryTool</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>memoryTool</name>
  <url>http://maven.apache.org</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.0.2</version>
        <configuration>
          <source>1.6</source>
          <target>1.6</target>
        </configuration>
      </plugin>
---------------------
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>exec-maven-plugin</artifactId>
        <configuration>
          <executable>java</executable>
          <arguments>
            <argument>-Xms512m</argument>
            <argument>-Xmx512m</argument>
            <argument>-XX:NewRatio=3</argument>
            <argument>-XX:+PrintGCTimeStamps</argument>
```

```
            <argument>-XX:+PrintGCDetails</argument>
            <argument>-Xloggc:gc.log</argument>
            <argument>-classpath</argument>
            <classpath/>
            <argument>com.redstack.App</argument>
          </arguments>
        </configuration>
      </plugin>
----------------------
    </plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Если вы предпочитаете не использовать Maven, то можете запустить приложение следующей командой:

```
java -Xms512m -Xmx512m -XX:NewRatio=3
  -XX:+PrintGCTimeStamps -XX:+PrintGCDetails
  -Xloggc:gc.log -classpath <whatever>
  com.redstack.App
```

При этом:
-Xms определяем исходный/минимальный размер heap в 512 мб
-Xmx определяем максимальный размер heap в 512 мб
-XX:NewRatio определяем размер old generation большим в три раза чем размер new generation
-XX:+PrintGCTimeStamps, -XX:+PrintGCDetails и -Xloggc:gc.log JVM печатает дополнительную информацию касаемо garbage collection в файл gc.log
-classpath определяем classpath
com.redstack.App main класс

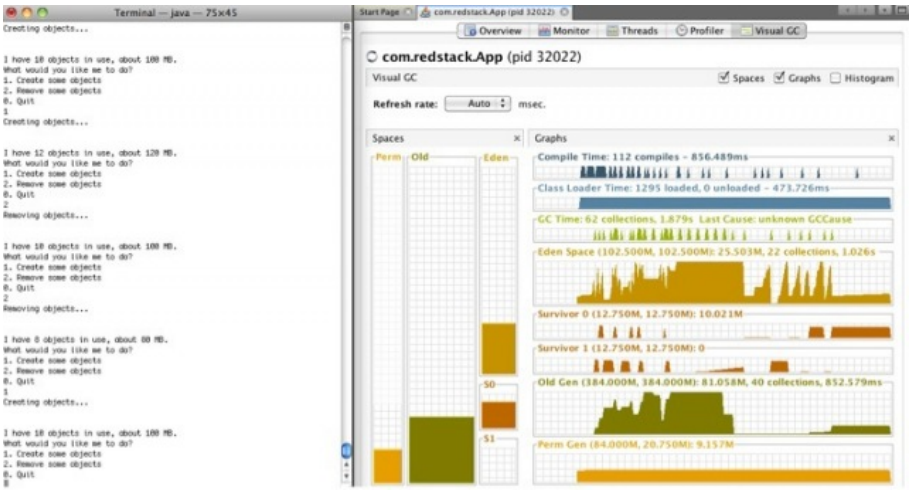
Ниже – код main-класса. Простая программа, в которой мы создаём объекты и далее выкидываем их, так что понятно, сколько памяти используется и мы можем посмотреть, что происходит с JVM.

```
package com.redstack;
import java.io.*;
import java.util.*;

public class App {

  private static List objects = new ArrayList();
  private static boolean cont = true;
  private static String input;
  private static BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

  public static void main(String[] args) throws Exception {
    System.out.println("Welcome to Memory Tool!");

    while (cont) {
      System.out.println(
        "\n\nI have " + objects.size() + " objects in use, about " +
        (objects.size() * 10) + " MB." +
        "\nWhat would you like me to do?\n" +
        "1. Create some objects\n" +
        "2. Remove some objects\n" +
        "0. Quit");
      input = in.readLine();
```

```
      if ((input != null) && (input.length() >= 1)) {
        if (input.startsWith("0")) cont = false;
        if (input.startsWith("1")) createObjects();
        if (input.startsWith("2")) removeObjects();
       }
     }

     System.out.println("Bye!");
   }

   private static void createObjects() {
     System.out.println("Creating objects...");
     for (int i = 0; i < 2; i++) {
        objects.add(new byte[10*1024*1024]);
      }
    }

    private static void removeObjects() {
      System.out.println("Removing objects...");
      int start = objects.size() - 1;
      int end = start - 2;
      for (int i = start; ((i >= 0) && (i > end)); i--) {
       objects.remove(i);
      }
    }
  }
```

Для сборки и выполнения кода используем следующую команду Maven:
mvn package exec:exec

Как только скомпилируете и будете готовы к дальнейшим действиям, запускайте ее и jvisualvm. Если вы не использовали jvisualvm ранее, то нужно установить плагин VisualGC: выберите Plugins в меню Tools, дальше вкладку Available Plugins. Выберите Visual GC и нажмите Install.
Вы должны увидеть список процессов JVM. Два раза кликните на том, в котором исполняется ваше приложение (в данном примере com.redstack.App) и откройте вкладку Visual GC. Должно появиться что-то типа того, что на скриншоте ниже.



Обратите внимание, что можно визуально наблюдать состояние permanent generation, old generation, eden и сегментов survivor (S0 и S1). Цветные колонки показывают используемую память. Справа находится historical view, которое показывает, когда JVM проводило garbage collections и количество памяти в каждом из сегментов.

В окне приложения начните создавать объекты (опция 1) и наблюдайте, что будет происходить в Visual GC. Обратите внимание на то, что новые объекты всегда создаются в eden. Теперь сделайте пару объектов ненужными (опция 2). Возможно, вы не увидите изменений в Visual GC. Это потому, что JVM не чистит это пространство до тех пор, пока не закончена процедура garbage collection.

Чтобы инициировать garbage collection, создайте ещё объектов, заполнив Eden. Обратите внимание, что произойдет в момент заполнения. Если в Eden много мусора, вы увидите, как объекты из Eden «переезжают» в survivor. Однако, если в Eden мало мусора, вы увидите, как объекты «переезжают» в old generation. Это случается тогда, когда объекты, которые необходимо оставить, больше чем survival.
Пронаблюдайте также за постепенным увеличением Permanent Generation.
Попробуйте заполнить Eden, но не до конца, потом выбросьте почти все объекты, оставьте только 20 мб. Получиться так, что Eden на большую часть заполнен мусором. После этого создайте ещё объектов. На этот раз вы увидите, что объекты из Eden переходят в Survivor.

А теперь давайте посмотрим, что будет, если у нас не хватит памяти. Создавайте объекты до тех пор, пока их не будет на 460 мб. Обратите внимание, что и Eden и Old Generation заполнены практически полностью. Создайте ещё пару объектов. Когда больше не останется памяти, приложение «упадёт» с исключением OutOfMemoryException. Вы уже могли сталкиваться с подобным поведением и думать, почему же это произошло – особенно, если у вас большой объем физической памяти на компьютере и вы удивлялись, как такое вообще могло произойти, что не хватает памяти – теперь вы знаете, почему. Если так случится, что Permanent Generation заполнится (довольно сложно в случае нашего примера добиться этого), у вас будет брошено другое исключение, сообщающее, что PermGen заполнен.

И, наконец, ещё один способ посмотреть, что происходило, это обратиться к логу. Вот немного из моего:

```
13.373: [GC 13.373: [ParNew: 96871K->11646K(118016K), 0.1215535 secs] 96871K->73088K(511232K), 0.1216535 secs] [Times
: user=0.11 sys=0.07, real=0.12 secs]
16.267: [GC 16.267: [ParNew: 111290K->11461K(118016K), 0.1581621 secs] 172732K->166597K(511232K), 0.1582428 secs] [Ti
mes: user=0.16 sys=0.08, real=0.16 secs]
19.177: [GC 19.177: [ParNew: 107162K->10546K(118016K), 0.1494799 secs] 262297K->257845K(511232K), 0.1495659 secs] [Ti
mes: user=0.15 sys=0.07, real=0.15 secs]
19.331: [GC [1 CMS-initial-mark: 247299K(393216K)] 268085K(511232K), 0.0007000 secs] [Times: user=0.00 sys=0.00, real
=0.00 secs]
19.332: [CMS-concurrent-mark-start]
19.355: [CMS-concurrent-mark: 0.023/0.023 secs] [Times: user=0.01 sys=0.01, real=0.02 secs]
19.355: [CMS-concurrent-preclean-start]
19.356: [CMS-concurrent-preclean: 0.001/0.001 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
19.356: [CMS-concurrent-abortable-preclean-start]
 CMS: abort preclean due to time 24.417: [CMS-concurrent-abortable-preclean: 0.050/5.061 secs] [Times: user=0.10 sys=
0.01, real=5.06 secs]
24.417: [GC[YG occupancy: 23579 K (118016 K)]24.417: [Rescan (parallel) , 0.0015049 secs]24.419: [weak refs processin
g, 0.0000064 secs] [1 CMS-remark: 247299K(393216K)] 270878K(511232K), 0.0016149 secs] [Times: user=0.00 sys=0.00, rea
l=0.00 secs]
24.419: [CMS-concurrent-sweep-start]
24.420: [CMS-concurrent-sweep: 0.001/0.001 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
24.420: [CMS-concurrent-reset-start]
24.422: [CMS-concurrent-reset: 0.002/0.002 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
24.711: [GC [1 CMS-initial-mark: 247298K(393216K)] 291358K(511232K), 0.0017944 secs] [Times: user=0.00 sys=0.00, real
=0.01 secs]
24.713: [CMS-concurrent-mark-start]
24.755: [CMS-concurrent-mark: 0.040/0.043 secs] [Times: user=0.08 sys=0.00, real=0.04 secs]
24.755: [CMS-concurrent-preclean-start]
24.756: [CMS-concurrent-preclean: 0.001/0.001 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
24.756: [CMS-concurrent-abortable-preclean-start]
25.882: [GC 25.882: [ParNew: 105499K->10319K(118016K), 0.1209086 secs] 352798K->329314K(511232K), 0.1209842 secs] [Ti
mes: user=0.12 sys=0.06, real=0.12 secs]
26.711: [CMS-concurrent-abortable-preclean: 0.018/1.955 secs] [Times: user=0.22 sys=0.06, real=1.95 secs]
26.711: [GC[YG occupancy: 72983 K (118016 K)]26.711: [Rescan (parallel) , 0.0008802 secs]26.712: [weak refs processin
g, 0.0000046 secs] [1 CMS-remark: 318994K(393216K)] 391978K(511232K), 0.0009480 secs] [Times: user=0.00 sys=0.00, rea
l=0.01 secs]
```

В логе можно увидеть, что происходило в JVM — обратите внимание, что использовался алгоритм Concurrent Mark Sweep Compact Collection algorithm (CMS), есть описание этапов и YG — Young Generation.

Можно воспользоваться этими настройками и в «продакшене». Есть даже инструменты, визуализирующие лог.

Java, JVM, garbage collection, garbage collector