# Java堆溢出

Java堆用于储存对象实例，我们只要不断地创建对象，并且保证GC Roots到对象之间有可达路径来避免垃圾回收机制清除这些对象，那么随着对象数量的增加，总容量触及最大堆的容量限制后就会产生内存溢出异常。

|  |
| --- |
| /\*\*  \* -Xms20M -Xmx20M -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/Users/deng4j/Downloads/packages/jvm.dump  \*  \* 限制Java堆的大小为20MB，不可扩展（将堆的最小值-Xms参数与最大值-Xmx参数设置为一样即可避免堆自动扩展），  \* 通过参数-XX：+HeapDumpOnOutOf-MemoryError可以让虚拟机在出现内存溢出异常的时候Dump出当前的内存堆转储快照以便进行事后分析。  \*/  public class HeapOOM {  public static void main(String[] args) {  List<HeapOOM> list =new ArrayList<>();  while (true){  list.add(new HeapOOM());  }  }  } |

|  |
| --- |
| java.lang.OutOfMemoryError: Java heap space  Dumping heap to /Users/deng4j/Downloads/packages/jvm.dump ...  Heap dump file created [33928434 bytes in 0.081 secs]  Exception in thread "main" java.lang.OutOfMemoryError: Java heap space |

要解决这个内存区域的异常，常规的处理方法是首先通过内存映像分析工具（如Eclipse Memory Analyzer或者intellij-idea Profiler）对Dump出来的堆转储快照进行分析。第一步首先应确认内存中导致OOM的对象是否是必要的，也就是要先分清楚到底是出现了内存泄漏（Memory Leak）还是内存溢出（Memory Overflow）。

# 虚拟机栈和本地方法栈溢出

由于HotSpot虚拟机中并不区分虚拟机栈和本地方法栈，因此对于HotSpot来说，-Xoss参数（设置本地方法栈大小）虽然存在，但实际上是没有任何效果的，栈容量只能由-Xss参数来设定。关于虚拟机栈和本地方法栈，在《Java虚拟机规范》中描述了两种异常：

* 如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出StackOverflowError异常。
* 如果虚拟机的栈内存允许动态扩展，当扩展栈容量无法申请到足够的内存时，将抛出OutOfMemoryError异常。

《Java虚拟机规范》明确允许Java虚拟机实现自行选择是否支持栈的动态扩展，而HotSpot虚拟机的选择是不支持扩展，所以除非在创建线程申请内存时就因无法获得足够内存而出现OutOfMemoryError异常，否则在线程运行时是不会因为扩展而导致内存溢出的，只会因为栈容量无法容纳新的栈帧而导致StackOverflowError异常。

|  |
| --- |
| /\*\*  \* -Xss208k  \* 设置栈内存208k  \*/  public class JavaVMStackSOF {  private int stackLength = 0;  public void stackLeak(){  stackLength++;  stackLeak();  }  public static void main(String[] args) {  JavaVMStackSOF javaVMStackSOF = new JavaVMStackSOF();  try {  javaVMStackSOF.stackLeak();  } catch (Throwable e) {  System.out.println("stack length:"+javaVMStackSOF.stackLength);  e.printStackTrace();  }  }  } |

|  |
| --- |
| stack length:828  java.lang.StackOverflowError |

|  |
| --- |
| /\*\*  \* -Xss208k  \* 设置栈内存208k  \*/  public class JavaVMStackSOF1 {  private static int stackLength = 0;  public static void test() {  long unused1, unused2, unused3, unused4, unused5, unused6, unused7, unused8, unused9, unused10, unused11, unused12, unused13, unused14, unused15, unused16, unused17, unused18, unused19, unused20, unused21, unused22, unused23, unused24, unused25, unused26, unused27, unused28, unused29, unused30, unused31, unused32, unused33, unused34, unused35, unused36, unused37, unused38, unused39, unused40, unused41, unused42, unused43, unused44, unused45, unused46, unused47, unused48, unused49, unused50, unused51, unused52, unused53, unused54, unused55, unused56, unused57, unused58, unused59, unused60, unused61, unused62, unused63, unused64, unused65, unused66, unused67, unused68, unused69, unused70, unused71, unused72, unused73, unused74, unused75, unused76, unused77, unused78, unused79, unused80, unused81, unused82, unused83, unused84, unused85, unused86, unused87, unused88, unused89, unused90, unused91, unused92, unused93, unused94, unused95, unused96, unused97, unused98, unused99, unused100;  stackLength++;  test();  unused1 = unused2 = unused3 = unused4 = unused5 = unused6 = unused7 = unused8 = unused9 = unused10 = unused11 = unused12 = unused13 = unused14 = unused15 = unused16 = unused17 = unused18 = unused19 = unused20 = unused21 = unused22 = unused23 = unused24 = unused25 = unused26 = unused27 = unused28 = unused29 = unused30 = unused31 = unused32 = unused33 = unused34 = unused35 = unused36 = unused37 = unused38 = unused39 = unused40 = unused41 = unused42 = unused43 = unused44 = unused45 = unused46 = unused47 = unused48 = unused49 = unused50 = unused51 = unused52 = unused53 = unused54 = unused55 = unused56 = unused57 = unused58 = unused59 = unused60 = unused61 = unused62 = unused63 = unused64 = unused65 = unused66 = unused67 = unused68 = unused69 = unused70 = unused71 = unused72 = unused73 = unused74 = unused75 = unused76 = unused77 = unused78 = unused79 = unused80 = unused81 = unused82 = unused83 = unused84 = unused85 = unused86 = unused87 = unused88 = unused89 = unused90 = unused91 = unused92 = unused93 = unused94 = unused95 = unused96 = unused97 = unused98 = unused99 = unused100 = 0;  }  public static void main(String[] args) {  try {  test();  } catch (Error e) {  System.out.println("stack length:" + stackLength);  throw e;  }  }  } |

|  |
| --- |
| stack length:54  Exception in thread "main" java.lang.StackOverflowError |

实验结果表明：无论是由于栈帧太大还是虚拟机栈容量太小，当新的栈帧内存无法分配的时候，HotSpot虚拟机抛出的都是StackOverflowError异常。

如果测试时不限于单线程，通过不断建立线程的方式，在HotSpot上也是可以产生内存溢出异常的。但是这样产生的内存溢出异常和栈空间是否足够并不存在任何直接的关系，主要取决于操作系统本身的内存使用状态。甚至可以说，在这种情况下，给每个线程的栈分配的内存越大，反而越容易产生内存溢出异常。

原因其实不难理解，操作系统分配给每个进程的内存是有限制的，譬如32位Windows的单个进程最大内存限制为2GB。HotSpot虚拟机提供了参数可以控制Java堆和方法区这两部分的内存的最大值，那剩余的内存即为2GB（操作系统限制）减去最大堆容量，再减去最大方法区容量，由于程序计数器消耗内存很小，可以忽略掉，如果把直接内存和虚拟机进程本身耗费的内存也去掉的话，剩下的内存 就由虚拟机栈和本地方法栈来分配了。因此为每个线程分配到的栈内存越大，可以建立的线程数量自然就越少，建立线程时就越容易把剩下的内存耗尽。

|  |
| --- |
| /\*\*  \* -Xss2M  \* 设置栈内存2M  \* <p>  \* 测试不断创建线程导致OOM，这很危险，会导致死机。  \*/  public class CreateThreadOOM {    public void stackLeakByThread() {  while (true) {  Thread thread = new Thread(new Runnable() {  @Override  public void run() {  dontStop();  }  });  thread.start();  }  }  private static void dontStop() {  while (true) ;  }  public static void main(String[] args) {  CreateThreadOOM sof = new CreateThreadOOM();  try {  sof.stackLeakByThread();  } catch (Error e) {  throw e;  }  }  } |

出现StackOverflowError异常时，会有明确错误堆栈可供分析，相对而言比较容易定位到问题所在。如果使用HotSpot虚拟机默认参数，栈深度在大多数情况下（因为每个方法压入栈的帧大小并不是一样的，所以只能说大多数情况下）到达1000~2000是完全没有问题，对于正常的方法调用（包括不能做尾递归优化的递归调用），这个深度应该完全够用了。但是，如果是建立过多线程导致的内存溢出，在不能减少线程数量或者更换64位虚拟机的情况下，就只能通过减少最大堆和减少栈容量来换取更多的线程。这种通过“减少内存”的手段来解决内存溢出的方式，如果没有这方面处理经验，一般比较难以想到，这一点需要在开发32位系统的多线程应用时注意。

# 方法区和运行时常量池溢出

由于运行时常量池是方法区的一部分，所以这两个区域的溢出测试可以放到一起进行。前面曾经提到HotSpot从JDK 7开始逐步“去永久代”的计划，并在JDK 8中完全使用元空间来代替永久代的背景故事，在此我们就以测试代码来观察一下，使用“永久代”还是“元空间”来实现方法区，对程序有什么实际的影响。

在JDK 6或更早之前的HotSpot虚拟机中，常量池都是分配在永久代中，我们可以通过-XX:PermSize和-XX:MaxPermSize限制永久代的大小，即可间接限制其中常量池的容量，以JDK 6来运行代码。

|  |
| --- |
| /\*\*  \* JDK 6  \* VM Args：-XX:PermSize=6M -XX:MaxPermSize=6M  \*  \* String::intern()是一个本地方法，它的作用是如果字符串常量池中已经包含一个等于此String对象的字符串，  \* 则返回代表池中这个字符串的String对象的引用；否则，会将此String对象包含的字符串添加到常量池中，并且返回此String对象的引用。  \*  \* 测试方法区常量池OOM  \*/  public class RuntimeConstantPoolOOM {  public static void main(String[] args) {  // 使用Set保持着常量池引用，避免Full GC回收常量池行为  Set<String> set = new HashSet<String>();  // 在short范围内足以让6MB的PermSize产生OOM了  short i = 0;  while (true) {  set.add(String.valueOf(i++).intern());  }  }  } |

|  |
| --- |
| Exception in thread "main" java.lang.OutOfMemoryError: PermGen space at java.lang.String.intern(Native Method) at org.fenixsoft.oom.RuntimeConstantPoolOOM.main(RuntimeConstantPoolOOM.java: 18) |

从运行结果中可以看到，运行时常量池溢出时，在OutOfMemoryError异常后面跟随的提示信息是“PermGen space”，说明运行时常量池的确是属于方法区（即JDK 6的HotSpot虚拟机中的永久代）的一部分。

而使用JDK 7或更高版本的JDK来运行这段程序并不会得到相同的结果，无论是在JDK 7中继续使用-XX：MaxPermSize参数或者在JDK 8及以上版本使用-XX：MaxMeta-spaceSize参数把方法区容量同样限制在6MB，也都不会重现JDK 6中的溢出异常，循环将一直进行下去，永不停歇。出现这种变化，是因为自JDK 7起，原本存放在永久代的字符串常量池被移至Java堆之中，所以在JDK 7及以上版本，限制方法区的容量对该测试用例来说是毫无意义的。

|  |
| --- |
| public class RuntimeConstantPoolOOM {  public static void main(String[] args) {  String str1 = new StringBuilder("计算机").append("软件").toString();  System.out.println(str1.intern() == str1);    String str2 = new StringBuilder("ja").append("va").toString();  System.out.println(str2.intern() == str2);  }  } |

在JDK 6中运行，会得到两个false，而在JDK 7中运行，会得到一个true和一个false。

产生差异的原因是，在JDK 6中，intern()方法会把首次遇到的字符串实例复制到永久代的字符串常量池中存储，返回的也是永久代里面这个字符串实例的引用，而由StringBuilder创建的字符串对象实例在Java堆上，所以必然不可能是同一个引用，结果将返回false。

而JDK 7（以及部分其他虚拟机，例如JRockit）的intern()方法实现就不需要再拷贝字符串的实例到永久代了，既然字符串常量池已经移到Java堆中，那只需要在常量池里记录一下首次出现的实例引用即可，因此intern()返回的引用和由StringBuilder创建的那个字符串实例就是同一个。而对str2比较返回false，这是因为“java”这个字符串在执行String-Builder.toString()之前就已经出现过了，字符串常量池中已经有它的引用，不符合intern()方法要求“首次遇到”的原则，“计算机软件”这个字符串则是首次出现的，因此结果返回true。

# 本机直接内存溢出

直接内存（Direct Memory）的容量大小可通过-XX：MaxDirectMemorySize参数来指定，如果不去指定，则默认与Java堆最大值（由-Xmx指定）一致。

|  |
| --- |
| /\*\*  \* VM Args：-Xmx20M -XX:MaxDirectMemorySize=10M  \*/  public class DirectMemoryOOM {  private static final int \_1MB = 1024 \* 1024;  public static void main(String[] args) throws Exception {  Field unsafeField = Unsafe.class.getDeclaredFields()[0];  unsafeField.setAccessible(true);  Unsafe unsafe = (Unsafe) unsafeField.get(null);  while (true) {  unsafe.allocateMemory(\_1MB);  }  }  } |

越过了DirectByteBuffer类直接通过反射获取Unsafe实例进行内存分配（Unsafe类的getUnsafe()方法指定只有引导类加载器才会返回实例，体现了设计者希望只有虚拟机标准类库里面的类才能使用Unsafe的功能，在JDK 10时才将Unsafe的部分功能通过VarHandle开放给外部使用），因为虽然使用DirectByteBuffer分配内存也会抛出内存溢出异常，但它抛出异常时并没有真正向操作系统申请分配内存，而是通过计算得知内存无法分配就会在代码里手动抛出溢出异常，真正申请分配内存的方法是Unsafe::allocateMemory()。

由直接内存导致的内存溢出，一个明显的特征是在Heap Dump文件中不会看见有什么明显的异常情况，如果读者发现内存溢出之后产生的Dump文件很小，而程序中又直接或间接使用了DirectMemory（典型的间接使用就是NIO），那就可以考虑重点检查一下直接内存方面的原因了。